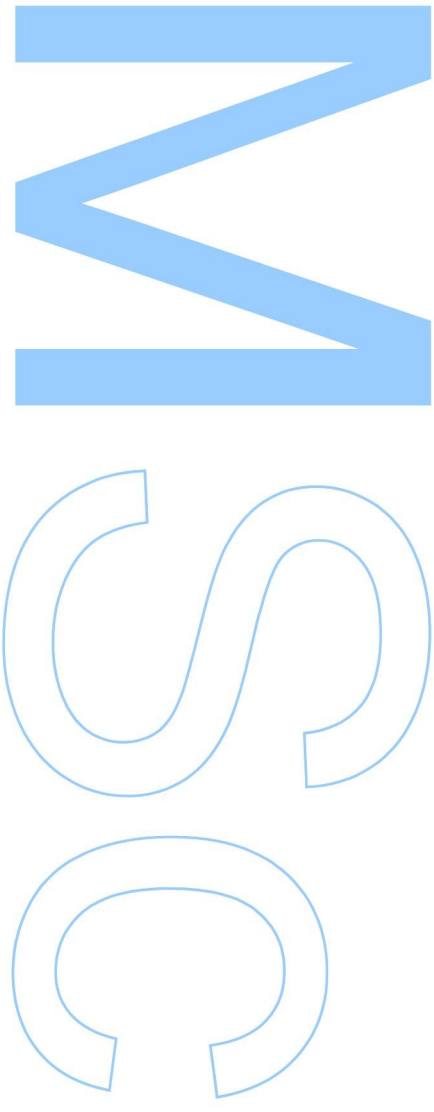
Improving the search for multi-relational concepts in ILP

Alberto José Rajão Barbosa
Mestrado em Ciência de Computadores
Departamento de Ciência de Computadores
2017

Orientador
Inês de Castro Dutra, Professor Auxiliar,
Faculdade de Ciências da Universidade do Porto

Coorientador
Pedro Manuel Pinto Ribeiro, Professor Auxiliar,
Faculdade de Ciências da Universidade do Porto



Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, / /

Abstract

Inductive Logic Programming (ILP) has been applied with success to many different domains, such as Medicine or Sciences, helping to generate interpretable first-order logic based models that can help to discriminate groups of interest. For example, in Medicine, an ILP model may help predicting and explaining why a tumor is malignant or benign. The downside is that ILP algorithms usually focus in searching for relations involving just one of the variables of the target concept to be learned. Learning concepts that relate multiple variables demand that the algorithms guide their search through the multiple paths that descend from each of those variables. There are few works on this direction in the literature, and ours is based on hyperpaths.

We present a new algorithm that runs the search over all the variable paths. Results show that searching through all variable paths renders theories that better explore the relations among the variables, translating to more meaningful knowledge extraction. With respect to accuracy, our theories perform equally or superior to the theories generated by a competitor ILP system. Our algorithm is however slower than our competitor, but we believe that improvements such as limiting the number of saturated examples, will bring execution times down and maintain the same level of accuracy.

Resumo

Programação Lógica Indutiva (PLI) tem sido aplicada com sucesso em diferentes domínios, como na Medicina ou na Ciência, auxiliando na geração de modelos baseados em lógica de primeira ordem facilmente interpretáveis que podem ajudar a discriminar determinados grupos de interesse. A desvantagem é que os algoritmos existentes em PLI normalmente apenas se focam em pesquisar relações envolvendo apenas uma das variáveis do conceito que se quer aprender. Aprender conceitos que relacionam múltiplas variáveis exige que os algoritmos guiem a sua pesquisa pelos múltiplos caminhos que descendem de cada uma dessas variáveis. Não existem muitos trabalhos nesta direcção na literatura, baseando-se o nosso em hipercaminhos.

Apresentamos um novo algoritmo que percorre todos os caminhos de todas as variáveis. Os resultados mostram que pesquisar por todos os caminhos das variáveis gera teorias que exploram melhor as relações entre as variáveis, traduzindo-se em extração de conhecimento mais significativo. Em relação à *accuracy*, as nossas teorias têm um desempenho melhor ou superior às teorias geradas por um sistema de PLI concorrente. O nosso algoritmo, no entanto, é mais lento que o concorrente, mas alguns melhoramentos, tais como limitar o número de exemplos a saturar, baixará os tempos de execução, mantendo o mesmo nível de *accuracy*.

Acknowledgments

I want to start by thanking my advisers Prof. Inês Dutra and Prof. Pedro Ribeiro for all the support through the realization of this thesis. Also, thank to all the elements of the project NanoStima for all the support and ideas. I would like to thank INESC TEC and again to project NanoStima for funding this investigation project. I also want to thank my family, in particular my parents and my brother for all the support in all moments, my girlfriend Inês for always believing in me, to my friends Duarte Petiz, Pedro Paredes and João Sousa for being pillars in my academic life. To finish, a general thank you to all my friends not mentioned specifically here, they were also important.

To my family.

Contents

List of Tables

List of Figures

List of Algorithms

Acronyms

ILP Inductive Logic Programming

wff Well-formed formula

RDB Relational Database

DHDB Deductive Hierarchical Databases

lub Least upper bound

glb Greatest lower bound

lgg Least general generalization

rlgg Relative least general generalization

HOC Head Output Clauses

SLD Selective Linear Definite

Chapter 1

Introduction

Inductive Logic Programming (ILP), over the past few years, has been successfully applied to many real world problems with a good amount of success. Some examples of these applications are predicting protein secondary structure [13] and predicting if a given breast tumor is benign or malignant [6].

Even with some success in many areas, there are still some areas in ILP that have not been explored properly, such as multi-relational concepts. Most ILP systems focus their search for new theories based on concepts involving only one variable (for example if a given train A goes east or not).

When it comes to learning concepts with more variables the algorithm should, ideally, explore all those variables and all the paths that arise from each of them so that it can be possible to learn, predict and get more accurate theories in multi-relational concepts (for example, how do we describe the relation of A being a grandparent of B).

In this document, we present an algorithm that searches the hypothesis space in a way that the final clauses, preferably connect all head variables through the literals they appear on. This way of searching the hypothesis space is based on the intuition presented on [29] that if there is a relation between variables, then that relation contains knowledge and so we should take advantage of clauses that contain such relations, since they contain knowledge.

Few works in the literature handle the problem of learning concepts that involve multiple variables. Ong et al. [26] show a hyperpath algorithm as a proof of concept to learn multi-variable concepts. Santos et al. [32] show an algorithm specialized for output variables that appear in the concept to be learned and focus on functional concepts like polynomials, factorial, among others. In our work we present a full implementation of the general problem of learning multiple-variable concepts with comparative results.

Besides presenting a novel algorithm for learning theories on multiple variable concepts, we also present a new evaluation metric that relates coverage and path length, in a way that clauses with good coverage and also with smaller paths of literals connecting the head variables in the

target concept will have better scores than other clauses. This evaluation metric is based on compression.

We experimented the implementation of our new algorithm with two different datasets. Results were compared with Aleph [35], a well-known ILP system, running on the same datasets. We evaluated the results looking at two different aspects: (1) is the new algorithm capable of producing more meaningful models than an algorithm that does not focus on connecting head variables? (2) are the more meaningful models more accurate than the ones that do not focus on connecting head variables? For the first set of experiments we compare the different models generated. For the second set, we used 10-fold cross validation to produce evaluation measurements. Except for being slower, our algorithm can compute more meaningful theories with equal or better performance than Aleph.

1.1 THESIS CONTRIBUTIONS

We created a new ILP system whose main search algorithm is based on hyperpaths. Our evaluation metric is based on compression. The implementation was done in C++, but we use Aleph's saturation and run it using the SWI-Prolog system.

1.2 THESIS OUTLINE

This chapter presented the motivation to this work, our objectives and contributions.

In Chapter 2 we present basic concepts and the theory of logic programming and inductive logic programming.

In Chapter 3 we present related work, summarise the contributions of other works and discuss about limitations of existing approaches.

In Chapter 4 we describe our design and implementation.

In Chapter 5 we present and discuss results of the approaches described in Chapter 4.

In Chapter 6 we summarize the major contributions of the thesis and present perspectives of future work.

Chapter 2

Background

In this chapter we present some theoretical concepts the reader must know to fully understand the rest of the document.

In Section 2.1 we introduce basic terminology and concepts in computational logic.

In Section 2.2 the reader can find a little introduction to ILP as well as some applications of ILP to some problems, as well as some theoretical aspects behind ILP systems, including some techniques and algorithms related to various stages of the learning process

In Section 2.4 we describe some existing ILP implementations.

In Section 2.5 we talk about some concepts related to graphs and hypergraphs that also relate to our work.

Most of the information in Section 2.2 is based on [9].

2.1 AN INTRODUCTION TO COMPUTATIONAL LOGIC AND LOGIC PROGRAMMING

This section is an essential part in understanding the concepts in the rest of the chapter, since here we will present some basics in computational logic that form the basis of more complex concepts in ILP.

We will approach four topics: Horn Clauses, SLD-Resolution, the entailment rule and the ground model. These are the concepts that form the core computational logic theory necessary to understand this work.

We keep our presentation short and do not present too many details. Throughout the text we give pointers and references to more complete sources.

2.1.1 Logic Programming Terminology

In this section we will describe some basic logic programming terminology and some deductive database terminology as well.

A first-order alphabet is composed by variables, predicate symbols and function symbols (constants included).

Definition 1. A variable is an upper case letter followed by a string of lower case letters and/or digits. ◊

Definition 2. A function symbol and a predicate symbol have the same representation: a lower case letter followed by a string of lower case letters and/or digits. ◊

Definition 3. A variable is a term. A function symbol immediately followed by a bracketed n-tuple of terms is a term. For example, *p(q(X),s,Y)* is a term, since p,q and s are function symbols and X and Y are variables. ◊

We define a constant to be a function symbol of arity 0.

Definition 4. A predicate symbol immediately followed by a bracketed n-tuple of terms is called an atom. L and its negation $\neg L$ are literals if L is an atom and we call L a positive literal and $\neg L$ a negative literal. ◊

We call a formula of the form $\forall X_1 \forall X_2 ... \forall X_q (L_1 \vee L_2 \vee ... L_s)$ a clause, where each X_i is a variable occurring in the terms L_j. We can also represent clauses as a finite set of literals, like $\{L_1, L_2, \neg L3, \neg L4\}$, and this set corresponds to the clause $(L1 \vee L2 \vee \neg L3 \vee \neg L4)$. This clause can be equally represented as $L1, L2 \leftarrow L3, L4$, where commas on the left side of $\leftarrow$ represent disjunctions and the commas on the right side of $\leftarrow$ represent conjunctions. A set of clauses represents the conjunction of clauses and it is called a clausal theory.

Literals, clauses and clausal theories are well-formed formulas (wff).

Definition 5. Let E be a wff or a term and $vars(E)$ the set of variables in E. E is ground if and only if $vars(E) = \emptyset$. ◊

We can also define a Horn clause as a clause that contains at most one positive literal and a definite program clause is a clause that contains exactly one positive literal. So we can represent it as $H \leftarrow L_1, ..., L_n$, where H and all the $L_i, i \in \{1, ..., n\}$ are atoms. More details on Horn clauses can be found in [11] and [36].

We call the positive atom in a definite program clause the *head*, and the conjunction of negative literals is called the *body* of the clause. A positive unit clause is a clause of the form $H \leftarrow$, which means that it has an empty body. In Prolog terminology, a general purpose logic programming language based in first-order logic that expresses the program logic in terms of relations among the concepts, we call that a fact, and denote it by H.

We define a definite logic program as a set of definite program clauses.

In Prolog, we can have literals in the form of $\neg L$, as long as L is an atom, in the body of the clause, where the $\neg$ is interpreted under the negation-as-failure rule [7].

Definition 6. A program clause is a clause of the form $H \leftarrow L_1, ..., L_n$ where T is an atom, and each of the $L_1, ..., L_n$ is of the form L or $\neg L$ and L is an atom.

So a program is a set of program clauses. ◊

A predicate definition is a set of program clauses with the same predicate symbol and arity in their heads.

We will now proceed with an example to illustrate some of the concepts described in this section so far.

Example 1. The clause $p(X, Y) \leftarrow s(X), q(Y, X)$ is a definite program clause, while $p(X, Y) \leftarrow \neg k(X), q(Y, X)$ is a program clause and the two constitute the definition of the predicate $p/2$, which is also a normal program.

Also we can state that the first is an abbreviated representation of $\forall X \forall Y : (p(X, Y) \vee \overline{s(X)} \vee \overline{q(Y, X)})$ that can be also written in set notation as $\{p(X, Y), \overline{s(X)}, \overline{q(Y, X)}\}$. ◄

We will now describe three types of restricted clauses, since these types of restrictions are often applied in many ILP systems.

Definition 7. A Datalog clause is a program clause where there can only be variables or constants in predicate arguments. ◊

Definition 8. A Constrained clause is a program clause where all variables in the body of the clause appear also in the head. ◊

Definition 9. A Non-recursive clause is a program clause in which the predicate symbol in the head does not appear in any literal of the body of the clause. ◊

Deductive databases are relational databases that allow both extensional, where every instance is explicitly described, and intensional, where a clause entails every possible instance of a given concept, definitions of relations and can be represented and implemented using logic programming. More on extensional and intensional descriptions will be brought up later. For that let us define a relation and then a relational database.

Definition 10. An n-ary relation p is a subset of the Cartesian product of n domains $D_1 \times D_2 \times ... \times D_n$, where a domain is a set of values. It is assumed that a relation is finite unless it is specified otherwise. ◊

A relational database (RDB) is a set of relations.

We can see a relation in an RDB as a predicate in a logic program.

Definition 11. A clause is typed if each variable that appears in arguments of the clause's literals is associated with a set of values that the variable can take. For example in $connected(X, Y)$, we may state that X and Y are of type person. The main difference between program clauses and database clauses is in the use of types. ◊

Types can increase the deductive efficiency by eliminating useless branches of the search space.

Definition 12. A database clause is a typed program clause of the form $H \leftarrow L_1, ..., L_n$ where H is an atom and $L_1, ..., L_n$ are literals. A deductive database is a set of database clauses. ◊

In database clauses there may be variables and function symbols in predicate arguments. This allows for recursive types and recursive predicates which makes the language much more expressive than the language of relational databases. If we restrict the recursiveness, we obtain a formalism for deductive hierarchical databases (DHDB), that are deductive databases where there are no recursive predicate definitions and no recursive types.

Non-recursive types determine finite sets of values which are constants or structured terms with constant arguments. Although DHDBs have the same expressive power as RDB, they allow intensional relations, which can be much more compact than RDB representations.

A predicate can be defined extensionally as a set of ground facts or intensionally as a set of database clauses, and it is assumed that each predicate is either described intensionally or extensionally. A relation in a database is essentially the same as a predicate definition in a logic program.

2.1.2 Selective Linear Definite (SLD) Resolution

SLD resolution is an inference rule. An inference rule is a logical form that takes premises and returns a conclusion (or more than one). An example of a inference rule, besides SLD resolution is *modus ponens* [33], which takes the premises *if p is true* and concludes that *"p then q" is true*, then q must be true. Inference rules preserve truth, in the sense that if the premises are true, so will the conclusion.

Since the SLD resolution algorithm uses variable substitutions, first, we introduce such concept.

Definition 13. A substitution $\theta = \{X_1/t_1, ..., X_n/t_n\}$ is a function from variables to terms. The application $W\theta$ of a substitution θ to a wff W is obtained by replacing all the occurrences of each X_i in W by the same term t_i. ◊

So we can define SLD resolution as: Given a goal clause $\neg l_1 \vee ... \vee \neg l_i \vee ... \vee \neg l_n$ and a selected literal $\neg l_i$ and an input definite clause $l \vee \neg k_1 \vee ... \vee \neg k_n$, whose positive atom l unifies with l_i, i.e., there is a substitution of variables that can turn the atom l into the atom l_i, SLD infers a

new goal clause where the selected literal is replaced by the negative literals of the input clause and the unifying substitution θ is applied, resulting in $(\neg l_1 \vee ... \vee \neg k_1 \vee ...\neg k_n \vee ... \vee \neg l_n)\theta$.

In SLD resolution, the resolution proof can be restricted to a linear sequence of clause of the kind $C_1, C_2, \dots C_n$, where C_1 is an input clause, and every other C_i is a resolvent (clause produced by applying resolution) one whose parent is the previous clause C_{i-1}. The proof is a refutation if C_n is the empty clause. Every clause in the sequence mentioned earlier is a goal clause, except for the first that is an input clause.

The selection function is responsible for choosing which literal will be replaced by a definite clause to perform another step in SLD resolution. Although in Prolog this selection function chooses literals by the order they are written, SLD resolution has no restrictions on which literal can be chosen at any time.

More information and details on SLD resolution can be found in [10] and [31].

This method is useful since it is often used when checking if a given hypothesis entails (c.f. next Section 2.1.3) a given example.

We will now present an example of the SLD-resolution procedure, that is a deductive procedure.

We will denote the resolvent of the clauses c and d as $res(c, d)$.

We will now address an example of a first-order derivation tree.

Example 2. Suppose that background knowledge B consists of $b_1 = female(mary)$ and $b_2 = parent(ann, mary)$ and $H = \{c\} = \{daughter(X, Y) \leftarrow female(X), parent(Y, X)\}$. Let $T = H \cup B$ and we want to derive the fact $e = daughter(mary, ann)$ from T. We do the following:

- We start by computing $c_1 = res(c, b_1)$ under the substitution $\theta_1 = \{X/mary\}$, which means we get $c\theta_1 = daughter(mary, Y) \leftarrow female(mary), parent(Y, mary)$. Then we resolve it with b_1 like in the propositional case, meaning that the resolvent of $daughter(X, Y) \leftarrow female(X), parent(Y, X)$ and $female(mary)$ is $c_1 = res(c, b_1) = daughter(mary, Y) \leftarrow parent(Y, mary)$.
- Then we need to do $c_2 = res(c_1, b_2)$ under the substitution $\theta_2 = \{Y/ann\}$. So $daughter(mary, Y) \leftarrow parent(Y, mary)$ and $parent(ann, mary)$ resolve in $c = res(c_1, b_2) = daughter(mary, ann)$.

We have the derivation tree in Figure 2.1.

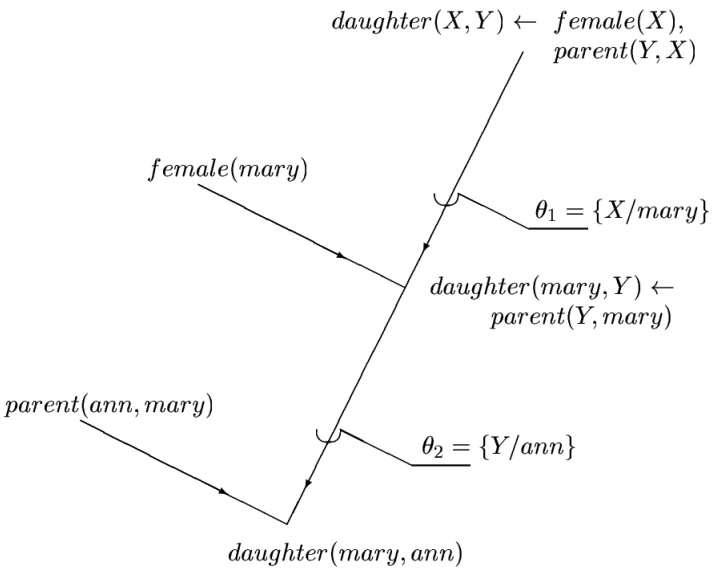


Figure 2.1: First-order derivation tree for SLD-resolution. Image taken from [9]

◀

2.1.3 Entailment

We can define logical entailment or logical consequence as a concept that evaluates if one statement logically follows from a set of other statements. We consider a logical argument as valid when the conclusions are entailed by the premises, since the conclusions are consequence of the premises.

We say, then, that a sentence s is the logical consequence of a set of sentences S, for some language, if and only if, the sentence s is true if every other sentence in the set S is also true. Transporting this to logic programming, we can say that statements are literals, and so a literal is the logical sequence of a set of literals (a clause) if and only if, the literal is true when every literal in the clause is also true.

This intuition is enough for the reader to fully understand the role of entailment in ILP, since this relation is explored when checking if a given example is explained by a given hypothesis.

If the reader wishes to proceed to a more detailed study on the entailment concept, we recommend [2].

2.1.4 Ground Model

In order to understand what a ground model in first-order languages is, we begin to define ground terms as terms that contain no variables, and we consider the Herbrand universe to be the set of all ground terms. A ground literal is an atomic formula whose argument terms are all ground terms, where the Herbrand base is the set of all ground literals and a Herbrand interpretation assigns a truth value to each ground literal in the base. A ground clause, on the other hand is a clause composed by ground literals. A ground model, in the context of our work, is the unique Herbrand base for a given definite clause theory T, $M^+(T)$, where the theory of the definite clause is transformed into a series of ground clauses over its Herbrand Universe as stated in Section 2.2.6.

For more information on Herbrand bases and Herbrand universes, please check [18].

2.1.5 Prolog syntax

In Prolog, the program logic is expressed as relations, being those relations constructed using terms, the single data type in Prolog. Relations are defined by clauses.

Although the single data type in Prolog is the term, it takes many different forms, namely atoms, numbers, variables and compound terms.

An atom is a general-purpose name with no inherent meaning. Examples of atoms are x and *'ball'*.

Numbers can be floats or integers.

Variables are denoted by a string of letters, numbers and underscore characters that begins with an upper case letter or an underscore.

A compound term is composed of an atom called the *functor* and an arbitrary number of *arguments* which are terms. These terms are written as a functor followed by a list of arguments separated by commas, contained in parentheses. An atom can, therefore, be seen as a functor with no arguments. Some examples of compound terms are *sides(square,4)* and *p(X,Y)*.

In Prolog we also have lists that are, essentially, an ordered collection of terms and are denoted by square brackets with the terms separated by commas. Examples of list are *[1,2,3,4]* and *[a,b,c,d]*.

Finally, Prolog also offers strings, that are sequences of characters surrounded by quotes and are equivalent to a list of numeric character codes. An example of a string is *"ILP is fun"*.

Prolog programs describe relations using clauses, more precisely Horn Clauses, and distinguish between two types of clauses: facts and rules.

A rule is of the form *Head :- Body* and means that the *Head* is true if the *Body* is true. The

body of the rule consists of calls to predicates that are the goals of the rule. The *,/2* connective means conjunction of predicate goals and the *;/2* means the disjunction of predicate goals. Both conjunctions and disjunctions can only appear in the body of the clause. An example of a rule is *p(X,Y) :- s(Y,Z), q(X).*

Clauses with empty bodies are called facts. An example of a fact is *p(X)* and can be written as the following rule *p(X) :- true*, where the *true* predicate is always true.

2.2 AN INTRODUCTION TO INDUCTIVE LOGIC PROGRAMMING

Inductive Logic Programming is roughly described as the intersection of machine learning and logic programming, which aims at learning logic programs inductively from examples.

2.2.1 The need of machine learning

We will begin to describe the importance of machine learning in ILP.

It is widely agreed that learning is crucial for any intelligent behavior, so the machine learning part of it is justified as part of the cognitive science. Machine learning techniques can also be used in knowledge acquisition tools, being generally accepted that the main problem in building intelligent systems is the acquisition of knowledge.

The goal of machine learning is to develop methods, techniques and tools for building intelligent learning systems that can be capable of changing themselves into performing better (in what concerns to efficiency, accuracy or addressing different kinds of problems) at certain tasks as they acquire different knowledge over the time.

Although we focus on inductive learning for its relation to Inductive Logic Programming, there are other machine learning paradigms such as deductive learning, genetic algorithms and connectionist algorithms (like neural networks). Michie [19] defines learning as the ability to obtain new knowledge, but requires that the result of learning has to be understandable by humans. This is a strong plus for ILP, as we will justify later, but removes neural networks and all the connectionist algorithms from the set of learning systems, since humans have high difficulties and sometimes even incapability when it comes to understanding the results of the learning process of such systems.

But we will now focus on inductive learning and state that it is generally known that induction means reasoning from specific to general. When we are learning inductively from examples, we generate theories and rules having as the basis of the learning concept those given examples.

This type of learning has been successfully applied to various areas such as the diagnosis of a patient's disease, where these problems are formulated into inductive concept learning problems, where classification rules for a given concept are induced from positive and negative instances of that concept.

It seems obvious that one can not speak of inductive concept learning without defining *concept*.

Definition 14. Let U be a universal set of *observations*. A concept C can be defined as a subset of U ($C \subseteq U$). Learning the concept C means learning to recognize objects $x \in C$ for each object $x \in U$. ◊

Example 3. As an example we can consider U as the set of all students in University of Porto and C the students that have a particular class. ◀

2.2.2 Description of observations and concepts

In the prior paragraphs we mentioned observations and we will now endorse this subject a little bit further, specially when it comes to their representations. To represent such observations we need an observation description language, and we can use the same language to describe concepts or use a different one. There are many widely used languages that describe observations and concepts, such as attribute-value, where each attribute of the object being observed has a value assigned to it, but we will use the first-order language of Horn clauses used in logic programming, where examples are described as ground facts.

Example 4. If we consider the game of billiards, we can describe the eight ball by *ball(8)*. ◀

When it comes to concept description, there are two ways to do it: *extensionally* and *intensionally*. A concept is described extensionally when we list the descriptions of all of its instances. This kind of description can carry many troubles such as the fact that a concept may contain an infinite number of instances (for examples, if we want to describe the parity of natural numbers). That being said, it is best to describe such concepts intensionally, in a different concept description language that can describe the same amount of objects in a more concise way, that is, in the form of rules.

Example 5. In billiards we can consider the following rule: *stripes(Ball) :- Ball > 8* to describe all the striped balls, instead of listing: *stripes(9), stripes(10), stripes(11), stripes(12), stripes(13), stripes(14), stripes(15).* ◀

It is obvious that describing concepts intensionally is way less costly than describing them extensionally. Notice that we are only purely taking into account the cost of representation and not considering the cost of generating these rules.

2.2.3 Inductive Concept Learning

Since we have already addressed the description of observations and concepts, we will now describe a definition that establishes if a given observation belongs to a given concept, i.e., if the description of the object satisfies the description of the concept.

If the observation belongs to the concept, we say the concept description covers the observation description.

From now on we will use the term *fact* to address an observation description and the term hypothesis for an intensional concept description that will be learned.

Definition 15. An example e for learning a concept C has a label ($\oplus$ if the observation is an instance of C and $\ominus$ otherwise). So if E is a set of examples, we denote E^+ as the set of positive examples (labeled with $\oplus$) and E^- as the set of negative examples (labeled with $\ominus$). ◊

Example 6. When we consider the learning concept $stripes(Ball)$ in the game of billiards, $stripes(4)$ is a negative example, while $stripes(10)$ is a positive example. ◄

We can then define the inductive concept learning problem as:

Definition 16. Given a set E of positive and negative examples of a concept C, find a hypothesis H, expressed in a given concept description language L such that:

- every positive example $e \in E^+$ is covered by H
- no negative example $e \in E^-$ is covered by H

◊

To implement coverage we need a function $covers(H, e)$ which returns true if e is covered by H and false otherwise. Since, in logic programming, a hypothesis is a set of program clauses and an example is a ground fact, SLD-resolution can be used to check whether e is entailed by H.

We say that a hypothesis H is complete with respect to E if $covers(H, E^+) = E^+$ and we say that H is consistent with respect to E if $covers(H, E^-) = \emptyset$.

So, according to the definition of concept learning problem we say that our hypothesis H must, if possible, be complete and consistent. Although ideally the hypothesis should be complete and consistent, in many real life cases it does not happen due to noise in the datasets and many other reasons.

2.2.4 Performance criteria of success

We said before that the goal of inductive concept learning is so that C and H agree in every example in E. But since one of the major aims of machine learning is to classify unseen examples, and there is absolutely no guarantee that H will correspond to C in new observations, we can see H as a classifier of new examples. So the main criteria of success of the learning system is its accuracy in classifying new observations.

That being said, we outline four performance criteria of success of learning systems:

- **Classification accuracy:** The accuracy of the hypothesis is measured by the percentage of new observations properly classified by it.
- **Transparency:** This means H's ability to be understandable by humans. There are many ways to measure it, such as the number of bits used in the encoding of the description or the length of the hypothesis (total number of conditions in the rule set H).
- **Statistical Significance:** Such test can be done to assure that H's performance represents a genuine regularity in the provided data, and is not accomplished just by chance.
- **Information Content:** Scales the classifier according to the difficulty of the classification problem and it is measured by taking into account the percentage of observations in U that belong to C. A correct classifier of a rare concept (that has way less observations in U) will provide much more information than a correct classifier of a more probable concept (that has much more observations in U).

2.2.5 Background Knowledge

If the learner has no previous knowledge about the problem, then it learns exclusively through examples. Nonetheless, the most difficult problems often require some specific previously known concepts (knowledge). Those concepts are called background knowledge, and using them, the learner can express the generalization of examples more naturally and concisely. The hypothesis language L and the background knowledge B define the search space of possible concept descriptions, called the hypothesis space. So we can define concept learning as searching the hypothesis space [20].

So now we can update some concepts stated above, such as completeness, consistency and the function *covers*.

The function *covers* is now defined as:

$covers(B, H, e) = covers(B \cup H, e)$

$covers(B, H, E) = covers(B \cup H, E)$

The completeness and consistency requirements are now updated to:

- **Completeness:** A hypothesis H is complete relatively to the background knowledge B and examples E if covers(B,H,E^+) = E^+.
- **Consistency:** A hypothesis H is consistent relatively to the background knowledge B and examples E if covers(B,H,E^-) = $\emptyset$.

By inserting background knowledge in our learning process, we can now change our definition of Inductive concept learning:

Definition 17. Given a set of training examples $E = (E^+ \cup E^-)$ and background knowledge B, find a hypothesis H, expressed in a concept language description L, such that H is complete and consistent regarding B and E. $\Diamond$

2.2.6 ILP Semantics

We will describe two different semantics in ILP: the normal and the nonmonotonic semantics.

2.2.6.1 Normal Semantics

We will suppose we are using a general setting for ILP that allows examples, background theory and hypothesis to be any well formed logical formula.

The problem of inductive inference is such that given the background knowledge B and examples E with positive examples E^+ and negative examples E^- we want to find a hypothesis H such that the following conditions are true.

Prior Satisfiabillity: $B \wedge E^- \not\models \square$

Posterior Satisfiability: $B \wedge H \wedge E^- \not\models \square$

Prior Necessity: $B \not\models E^+$

Posterior Sufficiency: $B \wedge H \models E^+$

The Posterior sufficiency criterion is also referred to as completeness and the Posterior satisfiabillity is the consistency.

In most ILP systems, background knowledge and hypothesis are restricted to definite clauses. There is a special case of the normal semantics, the definite setting, that is simpler than the general setting because a definite clause theory T has a unique minimal ground model $M^+(T)$ where any logical formula is either true or false in this model.

We formalize the setting as follows.

Prior Satisfiabillity: all $e \in E^-$ are false in $M^+(B)$

Posterior Satisfiabillity: all $e \in E^-$ are false in $M^+(B \wedge H)$

Prior Necessity: some $e \in E^+$ are false in $M^+(B)$

Posterior Sufficiency: all $e \in E^+$ are true in $M^+(B \wedge H)$

This is called the example setting and is equivalent to the normal semantics, where B and H are definite clauses and E is a set of ground unit clauses. The example setting is the main setting in ILP, being employed by most ILP systems.

The reason for allowing clausal evidence other than examples in the definite semantics is that

it is often useful to allow general clauses as evidence, since clausal evidence usually captures more knowledge than factual evidence consisting of only ground facts. The use of clausal evidence constrains the space of acceptable hypotheses, since future hypotheses have to also hold in the clauses that form the clausal evidence. Positive evidence has to be true in the ground model of the hypothesis and theory, and negative evidence has to be false in this setting.

2.2.6.2 The Nonmonotonic Semantics

In this setting of ILP, the background theory is a set of definite clauses, the evidence is empty and the hypothesis are sets of general clauses expressible using the same alphabet as the background theory. The evidence is empty since the positive part of it is considered part of the background theory and the negative part is derived implicitly, making a closed world assumption (using the ground model).

In this setting, the following conditions must hold for B and H:

Validity: all $h \in H$ are true in $M^+(B)$

Completeness: if general clause g is true in $M^+(B)$, then $H \models g$

Minimality: there is no proper subset G of H which is valid and complete.

The validity requirement assures that all clauses belonging to a hypothesis hold in the database B, which means that they are true properties of the data. The completeness requirement states that all information that is valid in the database should be encoded in the hypothesis and must be understood with regard to a given syntactic bias. The minimality requirement aims at deriving non redundant minimal hypothesis.

2.2.7 A generic ILP algorithm

Here we want to provide the reader with a general understanding of ILP algorithms and implementations.

We have to state that a key observation towards understanding this algorithm is to see ILP as a search problem.

Indeed, in ILP, there is a space of candidate solutions that is the set of well-formed hypothesis and an acceptance criterion characterizing solutions to an ILP problem. We could use an enumeration algorithm (generate and test), however this algorithm can be very expensive computationally to be interesting for practical use. So there is a need to structure the hypothesis space in order to prune some of the search space and this is made by means of the dual notions of generalization and specialization.

We can view induction as the inverse of deduction, if we look at generalization as induction and specialization as deduction.

Definition 18. We say that a hypothesis G is more general than a hypothesis S if and only if $G \models S$. S is more specific than G. ◊

We incorporate the notions of generalization and specialization using inductive and deductive inference rules.

Definition 19. A deductive rule $r \in R$ maps a conjunction of clauses G onto a conjunction of clauses S such that $G \models S$. Rule r is called a specialization rule. ◊

For example, adding a literal to a clause realizes specialization.

Definition 20. An inductive inference rule $r \in R$ maps a conjunction of clauses S onto a conjunction of clauses G such that $G \models S$. Rule r is called a generalization rule. ◊

An example of this rule is dropping a literal from a clause.

Generalization and specialization form the basis for pruning the search space, since

- When $B \wedge H \not\models e$, where B is the background knowledge, H is the hypothesis and e is a positive example, then none of the specializations H' of H will entail the evidence. They can, therefore, be pruned from the search.
- When $B \wedge H \wedge e \models \square$, where B is the background knowledge, H is the hypothesis and e is a negative example, then all generalizations H' of H will be inconsistent with $B \wedge E$.

Given the above ideas, we define the following generic ILP algorithm in Algorithm 1.

Algorithm 1 Generic ILP algorithm.

Input: Inference rules R **Output:** QH

1: $QH := Initialize$
2: **repeat**
3: Delete H from QH
4: Choose the inference rules $r_1, ..., r_k \in R$ to be applied to H
5: Apply the rules $r_1, ..., r_k$ to H to yield $H_1, ..., H_n$
6: Add $H_1, ..., H_n$ to QH
7: Prune QH
8: **until** stop-criterion(QH) satisfied

This algorithm keeps track of candidate hypotheses QH and repeatedly deletes a hypothesis H from the queue and expands that hypothesis using inference rules. These expanded hypotheses are added to QH that may be pruned to discard unpromising hypothesis from further consideration. This continues until the stop-criterion is satisfied.

We will explain, in some detail, the following generic parameters:

- *Initialize*, in line 1, denotes the hypothesis initial form. The initial hypothesis in ILP algorithms can take many forms and, generally, the form is related to the way the search through the hypothesis space is done. Algorithms that search the hypotheses space bottom-up, generalizing candidate hypotheses at each step, tend to choose more specific theories, like a positive example. On the other hand, algorithms that search the space of candidate hypotheses in a top-down manner, tend to choose more general initial theories, like the empty theory. So, we can see that the choice to be made in what concerns the initial hypothesis is highly related to the choice of inference rules, as we will see in the next points.

- R denotes the set of inference rules applied, that can be deductive, performing specialization, or inductive, performing generalization. This is, arguably, where the algorithms differ the most, since choosing specialization or generalization inference rules will lead to very different ways of solving the problems, as well as different advantages and disadvantages. Choosing specialization inference rules, will search the space of hypotheses using refinement operators, usually exploring some refinement data structure based on refinement graphs, which we will explore later. This means supporting, creating and, possibly, dynamically maintaining graph like data structures, which can involve many memory and time costs, in many cases, but, generally, accessing successors of a given candidate clause is much faster, since the operations to create new candidate clauses are, usually simpler, since they are, basically, adding a new literal to the clause or applying a variable substitution to it. On the other hand, generalization inference rules often employ much more complicated operators, in order to create new candidate theories, like relative least general generalizations or inverse resolution. On the other hand, they rely less on maintaining more complex data structures in order to generate new candidate clauses. This means that different kinds of inference rules lead to different needs and different advantages and disadvantages. All of this will be explained in future sections of this work specifically dedicated to studying the specialization and generalization techniques mentioned here.

- *Delete* influences the search strategy and using different strategies can lead one to perform depth-first, breadth-first or a best-first algorithm.

- *Choose* determines the inference rules to be applied on H. This can play a much more important role than it appears. Even though inference rules are either deductive or inductive, one can really apply some sort of mixed strategy, some times in different points of execution and so one has to choose correctly which action to apply to a given hypothesis, either to specialize it, or to generalize it. An example of different types of inference rules being used in different parts of an algorithm is Aleph where it takes an example (a very specific hypothesis) and generalizes it to a most-specific clause and then the algorithm proceed to taking the initial candidate clause as the empty clause and specializes this clause through the addition of literals and application of substitutions of variables.

- *Prune* determines which candidate hypothesis are to be deleted from the queue. This is usually done using probabilities or relying on the user. If one combines Delete and Prune, easily obtains advanced strategies such as hill-climbing, beam-search, etc. This pruning

techniques often involve applying scoring functions (heuristics) to candidate clauses, to determine which are, most likely, the best clauses to be eliminated. Such heuristics vary in many ways and depending on the goal one is seeking: for example, if we want an heuristic that will give us a clause that correctly classifies the highest number of already seen cases, we probably go for the coverage score function, that related the number of positive examples covered and the number of negative examples covered, however such scoring function can be very elementary and, for example, yield clauses that overfit the training data set, performing very well on those examples but poorly on unseen examples, so a compression like strategy, which relates coverage and the length of the clause, can be a good way to avoid clauses that are to specific, by bounding, in some way, the length of the candidate clauses. Other strategies for achieving different goals can be implemented, reflecting the quality and type of clauses one will get in the end of the algorithm. Another simple heuristic $f(c)$ is the expected accuracy of a clause $A(c) = \frac{TP+TN}{|E|}$, where TP stands for "True Positives", and it is the number of positive examples correctly classified as such by the theory, and TN stands for "True Negatives" and defines the number of negative examples correctly classified as such by the theory. Informativity, defined $I(c) = -log_2 p(\oplus \mid c)$ is another frequent heuristic. Another heuristics are accuracy gain $AG(c', c) = A(c') - A(c)$ and information gain $IG(c', c) = I(c) - I(c')$ as well as weighted accuracy gain $WAG(c', c)$ and weighted information gain $WIG(c', c)$.

- The *Stop-criterion* states the condition under which the algorithm stops. This is another aspect of an ILP algorithm where many systems differ. Stopping criteria can be related to many things, since the number of examples, to a possible threshold according to some scoring metric accomplished by a given theory, and many others. Again, we have a handful of strategies to explore. One can explore every existing example in the dataset, which will, probably, translate in a much more complete and thorough examination of such data set and much more reliable knowledge extraction, but, on the other hand, depending on the size of the dataset, can be a very costly approach, both in time and memory. Other approaches can be based on cover removal, that means that the positive examples already covered by the current hypothesis will not be analyzed by the algorithm. This can lead to a faster examination of the data, but also to a much more incomplete one and much more sensitive to noise. Other stopping criteria are applied in domains with non-noisy data and in domains with noisy data. In domains with perfect data, the stopping criterion requires consistency and completeness. When it comes to imperfect data, we apply heuristic stopping criteria. Both of these criteria are based on the number of positive and negative examples covered by a clause or hypothesis. Also there is a possibility, due to the big amounts of data in current data sets, that the stopping criterion could be related to the available resources in solving the problem, like time and memory of the computer.

2.2.8 Hypothesis Space structuring techniques

We can see concept learning as a search problem, where we consider concept descriptions the states in the hypothesis space and we want to find one or more states that satisfy some quality criterion. A learner can be described in terms of the structure of its hypothesis space, its search strategy and possibly search heuristics.

We will introduce now the concept of θ-subsumption lattice that provides the structure of the search space of program clauses. When the hypothesis space is structured the learner can search it either blindly, normally using a depth-first search or a breadth-first search, or heuristically, being the most common the hill-climbing or the beam search.

In ILP the hypothesis space is determined by the language of logic programs L, that consists of clauses of the form $H \leftarrow Q$, being H an atom $p(X_1, ..., X_n)$ and Q a conjunction of literals $L_1, ..., L_n$. The vocabulary of the background knowledge B will determine the vocabulary of the predicate symbols $q_i \in Q$ in the literals L_i of the clause.

The language bias L restricts syntactically the form of the clauses that can be formed from a given vocabulary of predicates, function symbols and constants of the language. In ILP the language bias is some restricted form of logic programs, for example, function-free program clauses as used in FOIL.

To be able to search this hypothesis space it is useful to establish a partial order in the set of possible clauses based on the θ-subsumption. We first need to recall the definition of substitution given earlier and then we will define θ-subsumption.

Definition 21. Let c and c' be two program clauses. $c\theta$-subsumes c' if there is a substitution θ such that $c\theta \subseteq c'$. Two clauses c and d are θ-subsumption equivalent if c θ-subsumes d and d θ-subsumes c. We say a clause is reduced if it is not θ-subsumption equivalent to any proper subset of itself. $\Diamond$

We will now use an example to illustrate the above notions.

Example 7. Let $c = daughter(X, Y) \leftarrow parent(Y, X)$. If we apply the substitution $\theta = \{X/mary, Y/ann\}$ to c, we obtain $c\theta = daughter(mary, ann) \leftarrow parent(ann, mary)$.

The clausal notation $daughter(X, Y) \leftarrow parent(X, Y)$ stands for $\{daughter(X, Y), \overline{parent(Y, X)}\}$, where all variables are universally quantified and the commas denote disjunction.

According to the definition of θ-subsumption we say that $c\theta$-subsumes c' if there is a substitution θ that can be applied to c such that every literal in $c\theta$ occurs in c'.

Clause c θ-subsumes $c' = daughter(X, Y) \leftarrow female(X), parent(Y, X)$ with $\theta = \emptyset$ since the set $\{daughter(X, Y), \overline{parent(Y, X)}\}$ is a subset of $\{daughter(X, Y), \overline{female(X)}, \overline{parent(Y, X)}\}$.

Clause c θ-subsumes $c' = daughter(X, X) \leftarrow female(X), parent(X, X)$ with $\theta = Y/X$

Clause c θ-subsumes $c' = daughter(mary, ann) \leftarrow female(mary),\ parent(ann, mary),\ parent(ann, tom)$ with $\theta = \{X/mary, Y/ann\}$.

Clauses $c = daughter(X, Y) \leftarrow parent(Y, X), parent(W, V)$ and $d = daughter(X, Y) \leftarrow parent(Y, X)$ are θ-subsumption equivalent, since they θ-subsume each other, and d is a reduced clause.

θ-subsumption introduces the syntactic notion of generality. Clause c is at least as general as clause c' if $c\theta$-subsumes c', so $c \leq c'$ and c is more general than c' if $c \leq c'$ holds but $c' \leq c$ does not hold and we say that $c < c'$. In this case, c' is a specialization of c and c is a generalization of c'. The only specializations considered usually by the learner are the most general specializations of the clause. ◀

We will now approach two properties of the θ-subsumption:

- If c θ-subsumes c' then c logically entails c', $c \models c'$. The reverse does not always hold.
- The relation $\leq$ introduces a lattice on the set of reduced clauses, meaning that any two clauses have a least upper bound (lub) and a greatest lower bound (glb), and they are both unique up to renaming of variables.

If we consider the second property of θ-subsumption, we get that the least general generalization (lgg) of two reduced clauses c and c' ($lgg(c, c')$) is the lub of c and c' in the θ-subsumption lattice.

Since θ-subsumption and least general generalization are purely syntactic notions, their computation is simple and easy to implement in any ILP system. The same is true for the notion of generality based on θ-subsumption. But if we take background knowledge into account we would deal with semantic generality: clause c is at least as general as c' with respect to background knowledge B if $B \cup c \models c'$. The syntactic generality is computationally much more feasible and semantic generality is generally undecidable and does not introduce a lattice on a set of clauses. So all this leads to syntactic generality being more frequently used in ILP systems.

θ-subsumption is important for the learning process for the following reasons:

- Provides a generality ordering for hypotheses, structuring the hypothesis space.
- Can be used to prune the search space.
  - When generalizing c to c', $c' < c$, all the examples covered by c will also be covered by c', since if $B \cup c \models e$ holds, then $B \cup c' \models e$ also holds. This property can prune the space search when looking for more general clauses when e is a negative example: if e is inconsistent, then all its generalizations will also be inconsistent. This means that the generalizations of c do not need to be considered.

- When specializing c to c', $c < c'$, an example not covered by c will not be covered by any of its specializations since if $B \cup c \not\models e$ is true then $B \cup c' \not\models e$ is also true. With this property we can prune the search of more specific clauses when e is an uncovered positive example: if c does not cover any positive example, none of its specializations will. So we do not need to consider the specializations of c.

θ-subsumption provides the basis of two important ILP techniques:

- bottom-up building of least general generalizations from training examples, relative to background knowledge, and
- top-down searching of refinement graphs.

2.2.9 Inference Rules in ILP

Given $B \wedge H \models E^+$, deriving E^+ from $B \wedge H$ is deduction and deriving H from B and E^+ is induction. So, inductive inference rules can be obtained by inverting deductive ones.

To control the application of inference rules, artificial intelligence employs operators that expand a given node in the search tree into a set of successor nodes in the search.

Definition 22. A specialization operator maps a conjunction of clauses G onto a set of maximal specializations of G. A maximal specialization S of G is a specialization of G such that G is not a specialization of S and there is no specialization S' of G such that S is a specialization of S'. ◊

Definition 23. A generalization operator maps a conjunction of clauses S onto a set of minimal generalizations of S. A minimal generalization G of S is a generalization of S such that S is not a generalization of G and there is no generalization G' of S such that G is a generalization of G'. ◊

One of these techniques is θ-subsumption. The other technique, that we will explore now, is inverse resolution.

Inductive inference rules can be viewed as the inverse of deductive rules of inference and since the deductive rule of resolution is complete for deduction, the inverse of resolution should be complete for induction.

We describe four rules of inverse resolution:

Absorption: $\frac{q \leftarrow A \quad p \leftarrow A,B}{q \leftarrow A \quad p \leftarrow q,B}$

Identification: $\frac{p \leftarrow A,B \quad p \leftarrow A,q}{q \leftarrow B \quad p \leftarrow A,q}$

Intra-Construction: $\frac{p \leftarrow A,B \qquad\qquad p \leftarrow A,C}{q \leftarrow B \quad p \leftarrow A,q \quad q \leftarrow C}$

Inter-Construction: $\frac{p \leftarrow A,B \qquad\qquad q \leftarrow A,C}{p \leftarrow r,B \quad r \leftarrow A \quad q \leftarrow r,C}$

In the previous rules, lower-case letters are atoms and upper-case letters are conjunctions of atoms. Absorption and Identification invert a single resolution step. This is shown in the image below where we design this operator as a "V", with the two premises in the base and one of the arms and the new clause is found in the other arm of the V. For this reason, we call Identification and Absorption the V-operators.

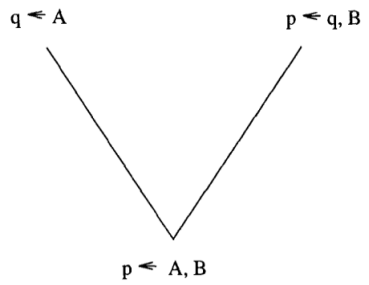


Figure 2.2: Example of a V-operator, in this case absorption. Image taken from [9].

The Inter- and Intra-Construction introduce a new predicate symbol, carrying out *predicate invention*. Diagrammatically, the construction operators can be shown as two linked Vs or a W, each representing a resolution. We place the premises at the two bases of the W and the three conclusions at the top of the W. One of the clauses is shared in both resolutions. Intra- and Inter-Construction are called the W-operators.

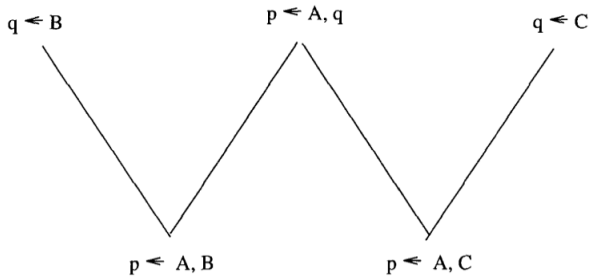


Figure 2.3: Example of a W-operator, in this case intra-construction. Image taken from [9].

2.2.10 Language Bias

A bias is a mechanism to constrain the search space for hypothesis and it can either determine how the hypothesis space is searched (search bias) or determine the hypothesis space itself (language bias).

By selecting a stronger language bias, the search space becomes smaller which makes learning more efficient. A stronger language bias means a less expressive language. Although it is good for

efficiency, a stronger bias can make the learner not find a solution due to too many constraints. So we have here a expressiveness/tractability trade-off.

But what does expressiveness mean in this context? We can evaluate it under two dimensions: the expressive power of the formalism itself or the length of the concept representation.

When it comes to the expressive power of the formalism, we mean that a stronger formalism can represent concepts that a weaker one can not. When it comes to the length, the best way to demonstrate it is with an example: in an attribute-value language we could represent that two boolean attributes have the same value with $([A = false] \wedge [B = false]) \vee ([A = true] \wedge [B = true])$. In a first-order language we could simply say that $A = B$.

Various different formalisms have been used to represent concepts and examples, ranging from propositional logic to first-order predicate calculus. Given that, one often distinguishes only between systems that learn attribute descriptions (attribute-value learners/propositional learners), from those which learn first-order relational descriptions (relation learners).

2.2.11 Predicate Invention

If P is a logic program, then the set of all predicate symbols found in the heads of the clauses of P is called the definitional vocabulary of P, $\mathcal{P}(P)$. ILP has three definitional vocabularies:

Observational vocabulary: $O = \mathcal{P}(E^+ \cup E^-)$

Theoretical vocabulary: $T = \mathcal{P}(B) - O$

Invented vocabulary: $I = \mathcal{P}(H) - (T \cup O)$

The learner carries out predicate invention when $I \neq \emptyset$.

Most predicate invention techniques are based on the use of W-operators. This involves a specific-to-general search.

2.2.12 Dealing with imperfect data

When learning concepts, it may happen that observation descriptions and/or their labels contain errors. In this cases, we are dealing with imperfect data. In this case, and in contrary to the definition stated in 2.2.3 it may be good not to require that H and C agree on all examples in E.

A very good characteristic of inductive learning systems is then the ability to avoid or minimize the effects of imperfect data by distinguishing between genuine regularities in the examples and regularities due chance.

The most frequent types of errors encountered are:

- noise (random errors in the background knowledge and/or examples)

- insufficiently covered example space, when the training example set is too sparse and it is difficult to extract correlations between the examples.
- inexactness, which means that the description language is inappropriate or insufficient to describe the target concept
- missing values in examples.

Learning systems often contain noise-handling mechanisms, whose mission is to prevent the induced hypothesis H from overfitting the data set E, i.e., to avoid the description of H to agree perfectly with C but poorly when it comes to unseen examples. These types of mechanisms often deal with the first three kinds of errors stated above. The last one generally is handled by a different mechanism.

When we deal with possibly imperfect data, we need to relax the consistency and completeness criteria defined in Section 2.2.3 and allow the hypothesis to classify incorrectly some of the examples in E.

2.2.13 Types of ILP systems

We can divide ILP systems in several categories. We have systems that can only learn a single concept and other systems that can learn multiple concepts. We have batch learners, that require all the examples to be given before the learning process and incremental learners, that accept examples along the learning process. Then, there are systems that rely on an oracle to verify the consistency of their generalizations and/or classify examples generated by the learner. Those are the interactive learners, otherwise are called non-interactive. Finally there are learners that learn concepts from scratch and learners that accept an initial theory and then revise it along the learning process (theory revisers).

Although all these characteristics are independent, there are two ways that most of the existing ILP systems stick to: batch non-interactive systems that learn single predicates from scratch (empirical ILP systems) and interactive and incremental theory revisers that learn multiple predicates (interactive/incremental ILP systems).

Interactive ILP systems typically learn multiple concepts from a small set of examples, relying on the answers of the user. An example of this kind of system is CIGOL [23].

Empirical ILP systems learn a single concept from a large set of examples. An example of this type of system is FOIL [28].

In Section 2.4 we will approach more characteristics of some ILP systems.

2.2.14 Mode Declarations

Here, we will explore an important part of our work, that are mode declarations, most-specific clauses and some definitions related to them. Since, in our approach we use most-specific clauses, also called bottom clauses, and they are generated through mode declarations, the reader must be familiar with some of the concepts described below. These concepts are present, in more detail in [32] and [35].

Definition 24. A mode declaration declare the mode of call for predicates that can appear in a clause and are of the form *mode(Recall,PredicateMode)*, where *Recall* bounds the non-determinancy of a form of predicate call and *PredicateMode* specifies a form of calling the predicate. *Recall* can be a number specifying the number of successful calls to the predicate or the symbol * specifying that the predicate is bounded for non-determinancy. *PredicateMode* is of the form *p(ModeType,ModeType, ...)*. An example of a mode declaration is *:- mode(*,has_car(+train,-car))*. *ModeType* can be simple or structured. If it is simple, $+T$ specifies that when a literal with predicate symbol p appear in a clause, the corresponding argument should be an input variable of type T, $-T$ specifies that the argument is an output variable of type T and $\#T$ declares that the argument is a constant of type T. Structures mode types are of the form *f(...)* being f a function symbol and each argument is a simple or structured ModeType. ◊

Definition 25. Let C be a definite clause with a total ordering over the literals and M be a set of mode declarations. $C = h \leftarrow b_1, ..., b_n$ is in the definite mode language $\mathcal{L}(M)$ if and only if h is the atom of a modeh declaration in M with every place-marker $+type$ and $-type$ replaced by variables and every place-marker $\#type$ replaced by a ground term and every atom b_i in the body of C is the atom of a modeb declaration in M with every place-marker $+type$ and $-type$ replaced by variables and every place-marker $\#type$ replaced by a ground term and every variable of $+type$ in any atom b_i is either of $+type$ in h or of $-type$ in some atom b_j with $1 \leq j < i$. ◊

Definition 26. Let C be a definite clause and v be a variable in C. Depth of v is defined as 0 if v is in the head of C or as $(\max_{u \in U_v} d(u)) + 1$ otherwise, where U_v are the variables in atoms in the body of C containing v. ◊

Definition 27. Let C be a definite clause with a defined total ordering over the literals and M be a set of mode declarations. C is in the depth-bounded mode language $L_i(M)$ if and only if C is in $L(M)$ and all variables in C have depth at most i according to the previous definition. ◊

It is possible, then, to form a bounded sub-lattice. This sub-lattice has a most general element, the empty clause ($\square$), and a least general element($\bot_i$) that is the most specific element in $L_i(M)$ such that $B \wedge \bot_i \wedge \overline{e} \vdash_h \square$, where $\vdash_h \square$ denotes derivations of the empty clause in at most h resolutions. We can, therefore describe a bottom clause $\bot_i$ for a depth-bounded mode language $\mathcal{L}_i(M)$ as the most specific definite clause in $L_i(M)$ that, along with the background knowledge B, entails a given example e.

2.2.15 Head Output Connected learning problems

Head Output Connected learning problems (HOC) is a class of learning problems where there is at least one output variable in the target concept.

They are a special case of the relational pathfinding [29].

The study of such clauses was useful to develop some work on our metric, as stated in more detail in Section 3.1.3.

Now we will introduce some definitions needed to understand better the HOC learning process, along with some examples.

Definition 28. A definite clause $h \leftarrow b_1, \ldots, b_n$ is *IO consistent* if and only if the input variables for each body atom b_i are either a subset of the head input variables or are found in a previous body atom b_j with $1 \leq j < i$. ◊

Example 8. Given the predicate signatures $c(+int)$, $a(+int, -int)$ and $b(+int, -int)$, we get that $c(X) \leftarrow a(X, Y), b(Y, Z)$ is IO consistent and $c(X) \leftarrow a(X, Y), b(Z, Y)$ is not. ◀

Definition 29. A definite clause C is HOC if and only if it is IO consistent and all its head output variables are instantiated in the body, existing a chain of literals connecting the head input variables to the head output variables. ◊

Example 9. Given the predicate signatures $c(+int, -int)$, $a(+int, -int)$ and $b(+int, -int)$, $c(X, Y) \leftarrow a(X, Z), b(Z, Y)$ is HOC since there is a chain of literals connection X to Y. ◀

Definition 30. A clause C' is a support clause (SC) for a clause C at depth D if and only if C' subsumes C, C' has D literals, C' is HOC and $SC(C') = C'$. ◊

This concept is very important in the context of head output variable connectedness. They are, at given depth, the shortest clauses that minimally connect the head input variables of the clause to its head output variables.

In the following example we show that the support clause at a given depth is not necessarily unique.

Example 10. Let $C = a(A, B) \leftarrow b(A, C), b(A, B), c(A, D), c(C, B), c(D, B)$ with the mode declarations $a(+t, -t), b(+t, -t), c(+t, -t)$ being t an arbitrary type, then:

- $SC_{C,1} = \{\{a(A, B) \leftarrow b(A, B)\}\}$
- $SC_{C,2} = \{\{a(A, B) \leftarrow b(A, C), c(C, B)\}\{a(A, B) \leftarrow b(A, D), c(D, B)\}\}$
- $SC_{C,3} = \{\}$

◀

HOC learning is robust to both determinate and non-determinate predicates both in the head or in the body of a clause, as shown in the previous example where b/2 and c/2 are non-determinate.

In order to search for the HOC clauses, we define an ordering of clauses from the Head to the most-specific clause, $\perp$.

Definition 31. Let $C = h \leftarrow b_1, \ldots, b_n$ be an arbitrary IO consistent clause. C' is a subclause of C if and only if it is of the form $h \leftarrow b'_1, \ldots, b'_k$ where $b'_1, \ldots, b'_k$ is a subsequence of $b_1, \ldots, b_n$ and C' is IO consistent. $\Diamond$

Example 11. If $C = c(X) \leftarrow a(X), b(X), d(X)$, $C' = c(X) \leftarrow a(X), b(X)$ and $C'' = c(X) \leftarrow b(X), a(X)$, then C' is a subclause of C but C'' is not a subclause of C, since its literals are not a subsequence of C. ◀

Definition 32. A clause C_{suc} is the successor of a clause C_{sub} with respect to a clause C if and only if C_{sub} is a subclause of C, C_{sub} is a subclause of C_{suc} and C_{suc} is one of the subclauses of C obtained by adding a single atom from the body of C to C_{sub}. The set of successors of a subclause C_{sub} with respect to a parent clause C ($S_{C_{sub},C}$) are all the C_{suc} clauses that can be generated from this definition. $\Diamond$

Example 12. Let $C = c(X,Y) \leftarrow a(X,A), b(A,B), d(A,D), e(D,Y)$ and $C_{sub} = c(X,Y) \leftarrow a(X,A)$, then $C_{suc_1} = c(X,Y) \leftarrow a(X,A), b(A,B)$ and $C_{suc_2} = c(X,Y) \leftarrow a(X,A), d(A,D)$. Then, $S_{C_{sub},C} = \{C_{suc_1}, C_{suc_2}\}$. $S_{C_{suc_1},C} = \{\}$ and
$S_{C_{suc_2},C} = \{c(X,Y) \leftarrow a(X,A), d(A,D), e(D,Y)\}$. ◀

2.2.16 Applications of ILP

In this section we will briefly describe some of the applications of ILP. There are three major areas of application of this kind of systems, that are knowledge acquisition, knowledge discovery in databases and scientific knowledge discovery. Nonetheless, there are so many other applications for ILP systems, such as logic program synthesis and inductive engineering, and some real life applications like predicting secondary structure of proteins [25]. We will describe in a little bit more detail the three classes of applications that are knowledge acquisition, knowledge discovery in databases and scientific knowledge discovery, since they can be seen as the basis of real life applications and play an important role in understanding some potential strengths and weaknesses of ILP systems.

2.2.16.1 Knowledge Acquisition

When building expert systems, the main concern is the acquisition of knowledge. The problem with it is that it takes too much time, since one has to consult specialists and often their knowledge can not be transformed in a way that make it suitable for machine use. This problem is called knowledge acquisition bottleneck and machine learning is contributing to widen this

bottleneck, developing tools that partially automate the process [5]. Newer inductive learning technology can be used to construct expert knowledge bases more efficiently than traditional techniques for knowledge acquisition. Currently, there are two trends in this subject.

On one side, ILP is concerned with using more powerful concept description languages, facilitating the use of domain-specific knowledge, since ILP can be used to construct knowledge bases automatically based on deep and qualitative models, and propositional learning techniques are not sufficient to do so. MOBAL [12] is an effective ILP knowledge acquisition tool.

On the other hand, there has been an investment in the development of toolkits of inductive learning techniques. It is known that it is difficult to solve the knowledge acquisition problem using only one technique and it is much more appropriate to have a toolkit of techniques and choose one or more of them according to the domain under investigation. So it was born the Machine Learning Toolbox (MLT) ESPRIT Project P2154 [15] that aimed at developing a workbench of machine learning tools and offering the chance to choose one or more that the user considers the most appropriate to the problem in hand. There is also the multistrategy learning paradigm that integrates two or more inference types and/or computational mechanisms. The LINUS [16] environment can be viewed as a toolkit of learning techniques similar to MLT.

In this application, empirical ILP is more appropriate for domains with a relatively well known domain knowledge in addition to a large number of examples of a single concept. When we are dealing with domains that require learning multiple concepts and little background knowledge is available, interactive ILP is the most appropriate way to go.

2.2.16.2 Knowledge Discovery in Databases

This application is directed to extracting non-trivial, implicit, previously unknown and potentially useful information from databases and one of main issues in this problem is the noise in data.

Some empirical ILP systems, such as FOIL [28] and GOLEM [24], already show the potential for knowledge extraction from large collections of data, and they are already efficient enough to be applied to real-life domains. But to discover interdependencies among different relations, multiple predicate learners should be applied to knowledge discovery task.

Inductive data engineering focuses on the inductive discovery of integrity constraints as well as on data base maintenance/design in accordance with such constraints.

2.2.16.3 Scientific Knowledge Discovery

There are many similarities between the discovery of scientific theories and construction of knowledge bases for expert systems. They usually both rely on a number of observations which are generalized using knowledge about the domain. This results in a new theory that can be considered a new piece of knowledge. This theory is a hypothesis that needs to be thoroughly

tested. All this can be aided by ILP, in the following phases:

- empirical or interactive generation of general logic theories from observations.
- interactive generation of experiments while discovering a logical theory.
- systematic testing of a given logical theory by trying to falsify it.

The first point can be made by any ILP system. GOLEM, back in the days, already generated theories meaningful for scientists [25]. The second point is addressed by interactive ILP systems and all experiments must be relevant and address different setting explained by the theory. The third point can also be performed by interactive ILP systems.

2.2.17 Empirical ILP

According to the definition in 2.2.5, we define inductive concept learning with background knowledge as follows: Given a set of examples E and background knowledge B, find a hypothesis H, expressed in a concept description language L, such that H is complete and consistent with respect to the background knowledge B and the examples E. Since we are talking about inductive logic programming, we define E as a set of ground facts, where the facts in E^+ are true and the facts in E^- are false in the intended context.

To make this definition feasible for ILP, we need to have the following notion of coverage, that affects the function covers, defined in 2.2.5: Given background knowledge B, hypothesis H and an example set E, H covers an example $e \in E$ with respect to background knowledge B if $B \cup H \models e$, i.e. $covers(B, H, e) = true$ if $B \cup H \models e$. So we define the function covers as: $covers(B, H, E) = \{e \in E \mid B \cup H \models e\}$.

Also remember the completeness and consistency requirements defined in 2.2.5.

The semantic notion of coverage states that e is covered by H given B if e is a logical consequence of $B \cup H$, denoted by $B \cup H \models e$, where $\models$ stands for semantic entailment and according to the definition of semantic entailment, $B \cup H \models e$ if $B \cup H \models_I e$ in every interpretation I, i.e., whenever an interpretation I makes $B \cup H$ true, it also makes e true.

This means that, in practice, for a given language bias L, we must use a procedure to check whether an example is entailed by $B \cup H$, being SLD-resolution the most often used, sometimes with a depth bound.

The notion of coverage described so far is called intensional coverage, since B is intensional and can contain ground facts and non-ground clauses. The extensional notion of coverage requires B to be in the form of ground facts only. Most empirical ILP systems use this extensional notion of coverage and interactive ILP systems use mostly the intensional one.

If B contains intensional predicate definitions on it, empirical ILP systems transform B into a ground h-easy model M of the background theory B that contains all true ground facts that

can be derived from B by a SLD-proof tree of depth less than h. Empirical ILP systems employ the function $covers_{ext}(M, H, e)$ (extensional cover), defined as follows:

Definition 33. A hypothesis H extensionally covers an example e with respect to a ground model M if there exists a clause $c = H \leftarrow Q$, and a substitution θ, such that $H\theta = e$ and $Q\theta = L_1, \ldots, L_n\theta \subseteq M$. ◊

We formulate empirical ILP, that learns a single predicate from a given set of examples, as follows:

Given:

- A set of training examples E, consisting of true E^+ and false E^- ground facts of an unknown predicate p,
- A description language L, that applies restrictions to the definition of p and
- Background Knowledge B, defining other predicates q_i that may be used in the definition of p and provide additional information about the arguments of the examples of p.

Find:

- A definition H for p, expressed in L, such that H is complete and consistent with respect to the examples E and background knowledge B.

We refer to the definition of p as the definition of the target relation.

When learning from noisy data sets, the completeness and consistency requirement must be relaxed to avoid overfitting.

Some empirical ILP systems are FOIL [28], mFOIL [8], GOLEM [24] and LINUS [16].

The complexity of learning is proportional to the expressiveness of the hypothesis language L, so, to reduce the complexity, there are some restrictions that can be applied to clauses expressed in L, for example, some ILP systems constrain L to function-free program clauses. Some other semantic bias include types in predicate arguments, input/output modes and parametrized languages, and many others.

2.2.18 Interactive ILP

We formalize interactive ILP, which is typically incremental as follows:

Given:

- A set of training examples E possibly with multiple predicates,

- Background knowledge B consisting of predicate definitions assumed to be correct.
- A current hypothesis H which is possibly incorrect,
- A description language L that applies restrictions to the definitions of predicates in H,
- An example e labeled $\oplus$ or $\ominus$ and
- An oracle that answers membership queries (labels examples as $\oplus$ and $\ominus$) and possibly other questions.

Find:

- A new hypothesis H', obtained by retracting and asserting clauses belonging to L from H such that H' is complete and consistent with respect to the examples $E \cup e$ and B.

Interactive ILP systems attempt to learn multiple relations that may be interdependent, from a small set of correct examples. During the learning process, interactive ILP systems create their own examples and ask the user about their label and may also ask about the validity of the generalizations constructed.

Several interactive ILP systems can perform a shift of bias, changing the hypothesis language L during the learning process. These changes include predicate invention and moving to a more powerful hypothesis language.

The space of clauses can be structured using refinement graphs.

Several interactive ILP systems are based on inverting the resolution principle. Most of the work done in this area is a special form of the general theory of inverse resolution introduced in CIGOL [23] that employs the three kinds of operators mentioned before: absorption, which generalizes clauses; intraconstruction, which introduces new auxiliary predicates; and identification which generalizes unit clauses.

2.3 ILP GENERAL TECHNIQUES

We can search the hypothesis space in two ways: bottom-up or top-down. With generalization techniques we search the hypothesis space bottom-up: we start from the training examples and search the hypothesis space by using generalization operators. Specialization techniques search the hypothesis space top-down, form the most general to specific concept descriptions, using specialization operators. Generalization techniques are best suited for interactive and incremental learning from few examples and specialization techniques are better suited for empirical learning, because we can apply heuristics to top-down search, dealing well with noisy data sets.

In this section we will talk about generalization and specialization techniques in more detail.

2.3.1 Generalization Techniques

Generalization techniques search the hypothesis space in a bottom-up manner. Bottom-up learners start from the most specific clause that covers a given example and then generalize the clause until it cannot further be generalized without covering negative examples.

A generalization c' of a clause c ($c' < c$) is obtained by applying a generalization operator based on θ-subsumption. Although we already defined a generalization operator earlier, we define it again for completeness purposes.

This generalization operator can be defined as:

Definition 34. Given a language bias L, a generalization operator ρ maps a clause c to a set of clauses $\rho(c)$ that are generalizations of c: $\rho(c) = \{c' \mid c' \in L, c' < c\}$ $\Diamond$

This operator performs two basic syntactic operations in a clause: first it applies an inverse substitution on the clause and then removes a literal from the body of the clause.

There are two basic classic generalization techniques: relative least general generalization 2.3.1.1 (used in GOLEM) and inverse resolution (used in CIGOL) and they are both used in hypothesis generation.

There is a unifying framework that covers both techniques, based on the notion of most specific inverse resolvent [21]. This unifying work was a first step in unifying theory of empirical and interactive ILP.

2.3.1.1 Relative Least General Generalization

The notion of least general generalization (lgg) forms the basis of cautious generalization algorithms that perform a bottom-up search of the θ-subsumption lattice. Cautious generalization assumes that if clauses c_1 and c_2 are true, it is very likely that $lgg(c_1, c_2)$ is also true. Recall that $lgg(c, c')$ is the least upper bound of c and c' in the θ-subsumption lattice. To compute the *lgg* of two clauses, we need to define the *lgg* of terms, atoms and literals first.

Definition 35.

- $lgg(t, t) = t$,
- $lgg(f(s_1, \ldots, s_n), f(t_1, \ldots, t_n)) = f(lgg(s_1, t_1), \ldots, lgg(s_n, t_n))$,
- $lgg(f(s_1, \ldots, s_m), g(t_1, \ldots, t_n)) = V$, where $f \neq g$ and V is a variable that represents $lgg(f(s_1, \ldots, s_m), g(t_1, \ldots, t_n))$,
- $lgg(s, t) = V$, where $s \neq t$ and at least one of s and t is a variable. In this case, V is a variable that represents $lgg(s, t)$.

◊

Some examples of the definition above are:

Example 13.

- $lgg([a,b,c],[a,c,d]) = [a,X,Y]$.
- $lgg(f(a,a),f(b,b)) = f(lgg(a,b),lgg(a,b)) = f(V,V)$ and V stands for $lgg(a,b)$.

◀

Definition 36.

- $lgg(p(s_1,\ldots,s_n),p(t_1,\ldots,t_n)) = p(lgg(s_1,t_1),\ldots,lgg(s_n,t_n))$ if atoms have the same predicate symbol p.
- $lgg(p(s_1,\ldots,s_m),q(t_1,\ldots,t_n))$ is undefined if $p \neq q$.

◊

Definition 37.

- if L_1 and L_2 are atoms (positive literals), then $lgg(L_1,L_2)$ is computed as defined above,
- if L_1 and L_2 are negative literals, $L_1 = \overline{A_1}$ and $L_2 = \overline{A_2}$, then $lgg(L_1,L_2) = lgg(\overline{A_1},\overline{A_2}) = \overline{lgg(A_1,A_2)}$,
- if L_1 is a positive literal and L_2 is a negative literal, or vice versa, then $lgg(L_1,L_2)$ is undefined.

◊

Example 14.

$lgg(parent(ann,mary),parent(ann,tom)) = parent(ann,X)$.

$lgg(parent(ann,mary),\overline{parent(ann,tom)}) = undefined$.

$lgg(parent(ann,X),daughter(mary,ann)) = undefined$. ◀

We can now define the least general generalization of two clauses in the following manner.

Definition 38. Let $c_1 = L_1,...,L_n$ and $c_2 = K_1,...,K_m$.
Then $lgg(c_1,c_2) = \{L_{ij} = lgg(L_i,K_j) \mid L_i \in c_1, K_j \in c_2$ and $lgg(L_i,K_j)$ is defined$\}$. ◊

Example 15. If $c_1 = daughter(mary, ann) \leftarrow female(mary), parent(ann, mary)$ and $c_2 = daughter(eve, tom) \leftarrow female(eve), parent(tom, eve)$, then $lgg(c_1, c_2) = daughter(X, Y) \leftarrow female(X), parent(Y, X)$, where X stands for $lgg(mary, eve)$ and Y stands for $lgg(ann, tom)$.

We define now the relative least general generalization (rlgg) of two clauses c_1 and c_2 as their least general generalization $lgg(c_1, c_2)$ relative to the background knowledge B. If the background knowledge consists of ground facts, and K denotes the conjunction of all these facts, the rlgg of two ground atoms A_1 and A_2 (positive examples), relative to background knowledge K is defined as: $rlgg(A_1, A_2) = lgg((A_1 \leftarrow K), (A_2 \leftarrow K))$. ◀

Example 16. Given the positive examples $e_1 = daughter(mary, ann)$ and $e_2 = daughter(eve, tom)$ and the background knowledge B: $parent(ann, mary)$, $parent(ann, tom)$, $parent(tom, eve)$, $parent(tom, ian)$, $female(ann)$, $female(mary)$, $female(eve)$. We compute the least general generalization of e_1 and e_2 relative to B as: $rlgg(e_1, e_2) = lgg((e_1 \leftarrow K), (e_2 \leftarrow K))$, where K denotes the conjunction of the literals $parent(ann, mary)$, $parent(ann, tom)$, $parent(tom, eve)$, $parent(tom, ian)$, $female(ann)$, $female(mary)$, $female(eve)$.

The following clause is generated by GOLEM $rlgg(e_1, e_2) = daughter(X, Y) \leftarrow female(X), parent(Y, X)$. ◀

2.3.1.2 Inverse Resolution

The basic idea of inverse resolution is to invert the SLD-resolution proof procedure, already approached before.

Inverse resolution (used in CIGOL [23], for example), inverts the resolution process using the generalization operator based on inverted substitution.

Definition 39. Given a wff W, an inverse substitution θ^{-1} of a substitution θ is a function that maps terms in $W\theta$ to variables, such that $W\theta\theta^{-1} = W$. ◊

Example 17. We will begin with an example of inverse substitution:

Let $c = daughter(X, Y) \leftarrow female(X), parent(Y, X)$ and $\theta = \{X/mary, Y/ann\}$:

$c' = c\theta = daughter(mary, ann) \leftarrow female(mary), parent(ann, mary)$.

If we apply the inverse substitution $\theta^{-1} = \{mary/X, ann/Y\}$ we get $daughter(X, Y) \leftarrow female(X), parent(Y, X) = c'\theta^{-1} = c$. ◀

Generally inverse substitution is quite more complex and involves places (represented by <>) of terms so we ensure that the variables in the initial wff W are restored in $W\theta\theta^{-1}$.

We will now exemplify inverse substitution with places.

Example 18. Let $W = loves(X, daughter(Y))$ and $\theta = \{X/ann, Y/ann\}$. So we get that $W\theta = loves(ann, daughter(ann))$. $\theta^{-1} = \{(ann, \{< 1 >(\text{first arugment of } loves)\})/X, (ann, \{<$

$2,1>$(second argument of loves and first subargument of *daughter*)$\})/Y\}$ specifies that the first occurrence (at place <1>) of the subterm ann in $W\theta$ is replaced by X and second occurrence (at place <2,1>) is replaced by Y. This ensures that $W\theta\theta^{-1} = loves(X, daughter(Y)) = W$. ◀

We will not elaborate theoretically the inverse resolution procedure, but will instead illustrate it with an example, where $ires(c, d)$ denotes the inverse resolvent of clauses c and d.

Example 19. Let B consist of $b_1 = female(mary)$ and $b_2 = parent(ann, mary)$, $H = \emptyset$ and the learner has a positive example $e_1 = daughter(mary, ann)$. The inverse resolution proceeds as follows:

- First, inverse resolution tries to find a clause c_1 that together with b_2 will entail e_1 and can be added to H instead of e_1. Choosing $\theta_2^{-1} = \{ann/Y\}$, the clause generated is $c_1 = ires(b_2, e_1) = daughter(mary, Y) \leftarrow parent(Y, mary)$ and $H = c_1$, such that $\{b_2\} \cup H \models e_1$. Note that, at this point, we already have a theory H that covers the only example provided (e_1), and for practical situations, this could be enough. However, we want to show that it is possible to obtain a more general theory using inverse resolution, so we show the next step in using this method.
- Then, we take $b_1 = female(mary)$ and $H = c_1 =$ $\{daughter(mary, Y) \leftarrow parent(Y, mary)\}$, we use the inverse substitution $\theta_1^{-1} = \{mary/X\}$, that generalizes c_1, originating $c' = daughter(X, Y) \leftarrow female(X), parent(Y, X)$. So, in H, c_1 can be replaced by c' which is a more general clause that together with B entails e_1. So the induced hypothesis is $H = \{daughter(X, Y) \leftarrow female(X), parent(Y, X)\}$.

Figure 2.4 presents the inverse derivation tree.

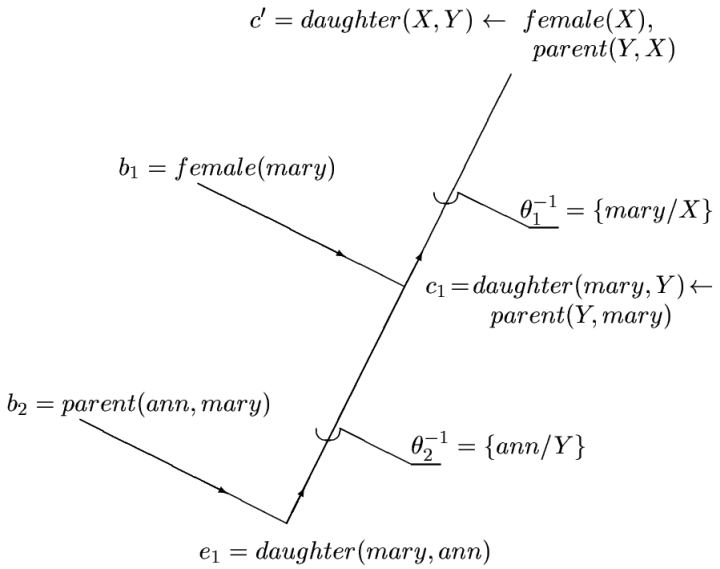


Figure 2.4: Inverse derivation tree. Image taken from [9].

◀

The generalization operator illustrated in the above example is the absorption (abduction) operator, presented in Section 2.2.9, as stated when presenting the inference rules of inverse resolution. It generates clauses using predicates that are not available in the initial vocabulary of the learner, i.e., using predicate invention.

2.3.1.3 Most specific inverse resolution

Like we have seen in the previous section, inverse resolution is non-deterministic, since at each step various generalizations can be made, depending on the choice of the clause to be inversely resolved and the employed inverse substitution.

We will demonstrate this with an example.

Example 20. Let us consider the example $e_1 = daughter(mary, ann)$ to be inversely resolved and B consists in $b_1 = female(mary)$ and $b_2 = parent(ann, mary)$.

- Let us suppose that b_2 is the first clause to be selected, since at each step various clauses from B can be selected.
- The question arises whether the generalization should be conservative (using an empty inverse substitution) or less cautious. In the last example, given $b_2 = parent(ann, mary)$ and $e_1 = daughter(mary, ann)$, the inverse substitution $\theta_2^{-1} = \{ann/Y\}$ was used to generalize e_1 to $c_1 = ires(b_2, e_1) = daughter(mary, Y) \leftarrow parent(Y, mary)$. We could

achieve a more cautious generalization using the most specific inverse substitution, in this case the empty inverse substitution $\theta_2^{-1} = \emptyset$ which would yield the most specific inverse resolvent of b_2 and e_1, $c_1 = ires_{ms}(b_2, e_1) = daughter(mary, ann) \leftarrow parent(ann, mary)$.

◀

To overcome this non-determinism problem, Muggleton [23] proposed that most specific inverse resolution would be used in generalization process, using most specific inverse substitution at each inverse resolution step. We will denote the most specific inverse resolvent of clauses c and d as $ires_{ms}(c, d)$.

We will now proceed with an example on this technique.

Example 21. Given $b_2 = parent(ann, mary)$ and $e_1 = daughter(mary, ann)$ using the empty substitution $\theta_2^{-1} = \emptyset$, we construct the most specific inverse resolvent $c_1 = ires_{ms}(b_2, e_1) = daughter(mary, ann) \leftarrow parent(ann, mary)$.

Then we generate $c' = ires_{ms}(b_1, c_1) =$
$daughter(mary, ann) \leftarrow female(mary), parent(ann, mary)$ using the empty substitution $\theta_1^{-1} = \emptyset$.

We present the most specific inverse linear derivation tree in Figure 2.5.

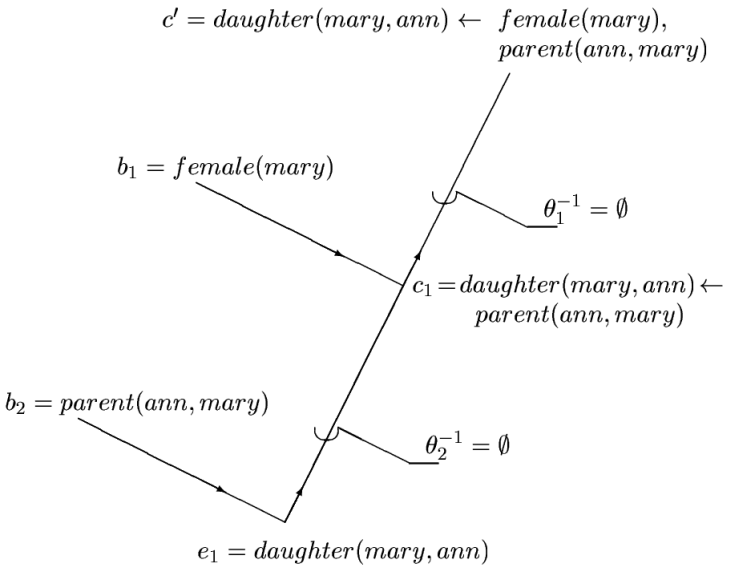


Figure 2.5: Most specific inverse linear derivation tree. Image taken from [9].

◀

This framework relates inverse resolution and relative least general generalization, in the

sense that relative least general generalizations are least general generalizations of most specific inverse linear derivations. We will illustrate this idea with an example.

Example 22. In this example we will assume that B consists of clauses $b_1 = female(mary)$, $b_2 = parent(ann, mary)$, $b_3 = female(eve)$ and $b_4 = parent(tom, eve)$. We also have two positive examples $e_1 = daughter(mary, ann)$ and $e_2 = daughter(eve, tom)$. We generate two most specific inverse linear derivation trees.

One of them is the same as the one in the last example, and the most specific inverse resolution results in the clause $c' = daughter(mary, ann) \leftarrow female(mary), parent(ann, mary)$.

The second one is generated in a similar way:

- We first inversely resolve e_2 with b_4 into $c_3 = ires_{ms}(b_4, e_2) = daughter(eve, tom) \leftarrow parent(tom, eve)$ with the empty substitution.
- Next we compute $c'' = ires_{ms}(b_3, c_3) =$ $daughter(eve, tom) \leftarrow female(eve), parent(tom, eve)$ under the empty substitution as well.
- Finally we compute c as the $lgg(c', c'')$, generalizing e_1 and e_2, yielding $c = daughter(X, Y) \leftarrow female(X), parent(Y, X)$.

We have a graphical description of this process in Figure 2.6

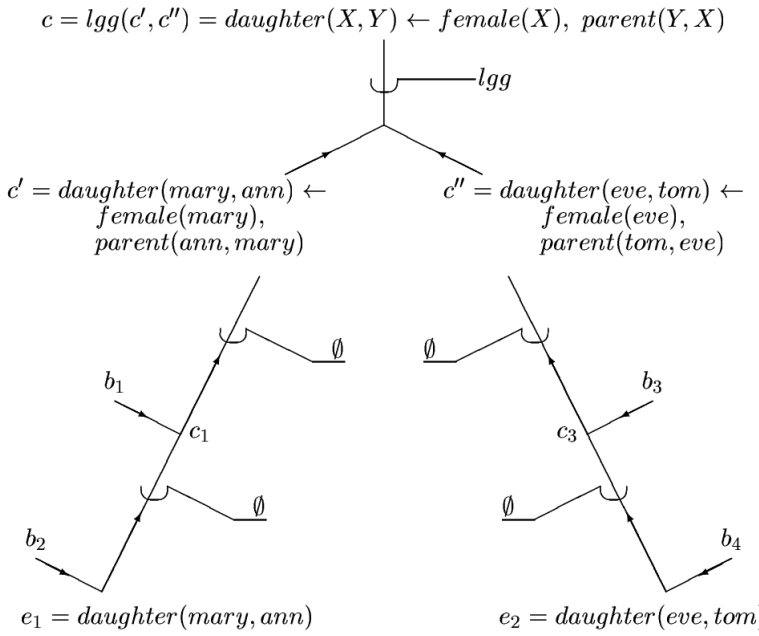


Figure 2.6: Graphical description of the process described above. Image taken from [9].

◀

Here we only describe the basic idea of this unifying framework for bottom-up generalization methods, where relative least general generalizations are least general generalizations of most specific inverse linear derivations. A more detailed explanation can be found in [21].

2.3.2 Specialization Techniques

While generalization techniques perform a bottom-up search in the hypothesis space, specialization techniques perform a top-down search from general to specific hypothesis, using a θ-subsumption based specialization operator called the refinement operator.

Definition 40. Given a language bias L, a refinement operator ρ maps a clause c to a set of clauses $\rho(c)$ that are refinements of c, such that $\rho(c) = \{c' \mid c' \in L, c < c'\}$. ◊

Usually this operator only computes the set of most general specializations of a clause under θ-subsumption, and employs two basic steps: applying a substitution to the clause and adding a literal to the body of the clause.

The basic specialization technique in ILP is top-down search of refinement graphs, where learners start from the most general clauses and refine them until they no longer cover negative examples and ensure that each refined clause covers at least one positive example.

This is the main technique used in empirical ILP systems.

2.3.2.1 Top-down search of refinement graphs

For a given language bias L and a given background knowledge B, the hypothesis space of program clauses is a lattice, structured by the θ-subsumption ordering.

We can define a refinement graph and use it to direct the search from most general to specific hypothesis.

Definition 41. We define a refinement graph to be a directed, acyclic graph in which nodes are program clauses and arcs correspond to the basic refinement operations: substitute a variable with a term and add a literal to the body of a clause. ◊

A simple algorithm that works based on this structure is the MIS (Model Inference System created by Ehud Shapiro) algorithm [34].

We will not present and explain the algorithm in detail, but, instead, present an example that focus, essentially, the use of refinement graphs in such algorithm, presenting to the reader a practical, yet simple, example on this topic.

Example 23. We restrict L to non-recursive definite clauses and the specialization operation used is adding a literal to the body of the clause.

Let us assume that the first example given to the learner is the positive example $e_1 = daughter(mary, ann)$. The top-level node in the refinement graph is the most general clause false (denoted by $\square$ or the empty clause $\leftarrow$), formally. In practice we start the search with the most general definition of the predicate daughter: $c = daughter(X, Y) \leftarrow$, where the empty body represents the body true. Since c covers e_1, we initialize $H = c$.

Then we encounter another positive example $e_2 = daughter(eve, tom)$ and H also covers it, so we do not need to change the hypothesis.

The third example we find is a negative one $e_3 = daughter(tom, ann)$. So the learner deletes c from H, because it covers a negative example and enters the second if to generate a new clause that covers e_1. Now the learner generates the set of refinements of c of the form $\rho(c) = daughter(X, Y) \leftarrow L$ being L one of following literals:

- Literals that have as arguments the variables in the head of the clause: $X = Y$, $female(X)$, $female(Y)$, $parent(X, Y)$, $parent(Y, X)$, $parent(X, X)$ and $parent(Y, Y)$.
- Literals that introduce a new variable Z ($Z \neq X$ and $Z \neq Y$): $parent(X, Z)$, $parent(Z, X)$, $parent(Y, Z)$ and $parent(Z, Y)$.

We present in Figure 2.7 part of the refinement graph.

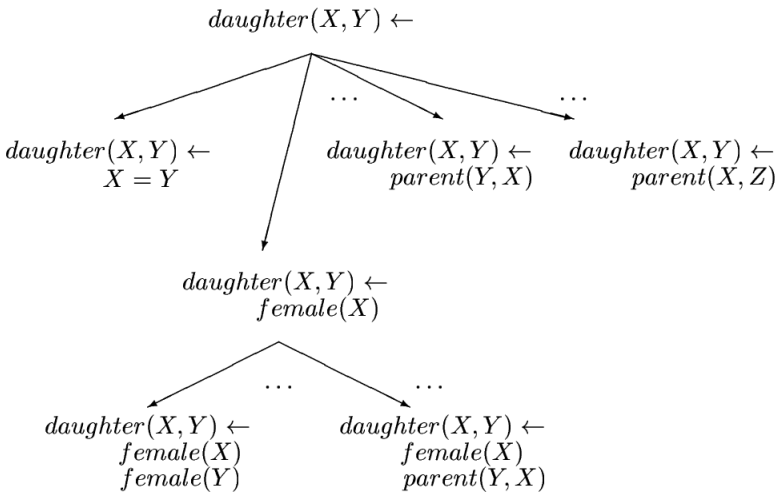


Figure 2.7: Refinement graph of the example. Image taken from [9].

We then consider the refinements one by one. The first to be considered is $daughter(X, Y) \leftarrow X = Y$ but it does not cover e_1 so it is retracted. Then we consider the refinement $c' = daughter(X, Y) \leftarrow female(X)$ and we find that it discriminates between positive and negative examples and it is retained in the current hypothesis H. If there were no more examples this would be the result of the learning process.

But let us consider that we found another negative $e_4 = daughter(eve, ann)$ that turn c' inconsistent, and we delete it from H turning it into the empty set again.

We repeat the clause generation process and the system tries to build a clause that covers e_1 and we consider the remaining refinements of c. $daughter(X, Y) \leftarrow female(Y)$ and $daughter(X, Y) \leftarrow parent(Y, X)$ are retained in the queue since they cover e_1, but are not consistent. Finally we consider the refinements of c' and among them there is $daughter(X, Y) \leftarrow female(X), parent(Y, X)$ that covers e_1 and is consistent. It also covers e_2 so the learner stops the search for new clauses.

◀

2.4 ILP SYSTEMS

In this section we will describe some existing ILP systems.

The list of systems that will be briefly described are: FOIL, FOCL, FORTE, GOLEM, HYDRA, LINUS, MFOIL, PROGOL, TILDE and Aleph.

Finally we will compare them all based on some characteristics important to our work.

2.4.1 FOIL

FOIL [28] is one of the best-known and successful empirical ILP systems and inspired a lot of research and learns intensional concept descriptions from relational tuples.

It induces concept descriptions as function-free Horn Clauses and represents background knowledge as sets of ground tuples (extensionally).

The general idea of its search algorithm is inducing a single clause starting with a clause with an empty body that is specialized by adding literals to its body. The candidates to be added to the body of the clause are constructed by generating all combinations of predicate names and variables such that at least one of those variables in the candidate exists in the current clause. Then it takes into account literals that state the equality or inequality of variables in the head or body literals and the literals may contain constants that the user has declared as theory constants. All literals have to conform to the type restrictions of the predicates.

For a more complete control of the language bias, FOIL provides limitation in the number and depth of variables in a single clause and incorporates mechanisms that exclude literals that lead to endless loops in recursive hypothesis clauses.

Among the candidate literals, FOIL selects the "best" according to the information gain heuristic.

FOIL stops adding literals to the clause if a predefined minimum accuracy is reached or

if the encoded length of the clause exceeds the number of bits needed for explicitly encoding the positive examples it covers (so it does not induce overly long and specific clauses in noisy domains).

Induction of further hypothesis stops if all positive examples are covered or if the set of induced hypothesis violates the encoding length restriction.

When post-processing clauses, FOIL removes unnecessary literals from induced clauses and redundant clauses from the concept definition.

The greedy strategy used by FOIL makes it very efficient, but also probable to exclude some good concept definitions from the hypothesis space. Some refinements to the hill-climbing search strategy augment its shortsightedness.

The examples may be provided by the user or the system may generate the negative examples using the closed world assumption.

Examples and background knowledge in FOIL are represented as tuples where each ground instance of a predicate is represented as a sequence of argument values. For each predicate, the user provides a header that defines its name and the types of the arguments. Input/output modes can also be used by the user limiting the number of candidate body literals. The user may also present some test cases to FOIL that then checks the hypothesis on these cases and reports the results.

2.4.2 FOCL

FOCL [27] learns Horn clause programs from examples and optional background knowledge. It integrates explanation based learning with the learning process of FOIL. In a nutshell, explanation based learning (EBL) exploits a very strong domain theory so it can, therefore, assume some generalizations and form new concepts from examples. We can define, roughly, a domain theory as the axioms about a certain domain of interest, i.e., background knowledge. With FOCL one can define intensionally background knowledge and it accepts a partial, possibly incorrect rule as input as an approximation of the target predicate. It also allows the use of user-defined constraints that reduce the search space.

2.4.3 FORTE

FORTE [30] stands for First Order Revision of Theories from Examples, and its name says a lot about the system itself, being a theory reviser. It is a system for revising automatically function-free first-order Horn clause theories. It has a set of specialization and generalization operators and uses a hill-climbing search through the hypothesis space. Each iteration generates all possibilities for applying each of its operators and the operation that generates a theory with the best accuracy is performed. This process continues while it is possible to increase the

accuracy of the theory or reduce its size.

2.4.4 GOLEM

GOLEM [24] is an empirical ILP system, as well as FOIL and is quite efficient with large data sets because it does not search the hypothesis space for consistent hypothesis, as FOIL does, but constructs a clause that covers a set of positive examples relative to the available background knowledge, using the principle of relative least general generalizations, mentioned in 2.3.1. GOLEM incorporates the rlgg construction in a covering approach.

To induce a single clause, it selects random pairs of positive examples and computes their rlggs and chooses the one that covers the most positive examples and is consistent. Then, it generalizes this clause. In the next step GOLEM selects again a set of examples, adds the clause obtained in the previous step, and computes their rlggs and chooses the one that covers the most positive examples and is consistent, as in the previous step. When the coverage of the best clause stops increasing, it stops and begins a post-processing step where the irrelevant literals are removed.

Since the rlgg may contain infinite literals, GOLEM, to ensure that the length of rlggs grows polynomially in the worst case with the number of examples, imposes that the background knowledge has to consist of ground facts, that the hypothesis language applies the determinacy restriction (if one gives values to the head variables of the clause, the values of the arguments on its body are unique) and the complexity of the hypothesis language is controlled by two parameters, i and j, that respectively limit the number and depth of body variables in a hypothesis clause.

GOLEM learns Horn clauses with functors and can be run in an interactive way. It is able to learn only from positive examples, since negative examples are only used in clause reduction, in the post-processing step and in the input/output mode declarations. When it comes to noisy data, GOLEM enables the user to define the maximum number of negative examples a hypothesis clause can cover.

2.4.5 HYDRA

HYDRA [1] extends the FOCL system adding likelihood ratios to the induced classification rules, learning a concept description for each class. The generated descriptions compete to classify test examples based on the likelihood ratios associated to clauses of that concept description, making the algorithm more robust towards noise.

2.4.6 LINUS

LINUS [9] is an ILP learner that incorporates different attribute-value learning systems, namely ASSISTANT and NEWGEM, in the sense that one can access other systems from within LINUS.

An attribute-value learning system is, in a nutshell, a system that represents knowledge using an attribute-value representation, where the examples are described in a table, where each attribute of interest of the example is described in a single column and each row corresponds to a single example. This is one possible way to represent knowledge and is applicable to many problems [39].

The main idea in LINUS is to transform a task of learning relational descriptions into an attribute-value learning task. It transforms a restricted class of ILP problems into propositional form and then solves it with an attribute-value algorithm. Then, it re-transforms the result into the first-order language. With this idea it can apply successfully propositional learners in a first-order framework. Since we can access various propositional learners in LINUS, it can also be seen as an ILP toolkit. It can run in CLASS or RELATION mode. In CLASS mode it is an enhanced attribute-value learner, in RELATION mode it is an ILP system.

In the transformation phase, the main idea is that all body literals that may appear in a hypothesis clause are determined, taking into account variable types. Each of the body literals correspond to a boolean attribute in the propositional formalism. Each example is transformed into attribute-value tuples. Taking into account the types of the argument variables of the examples, all possible applications of the background predicates on the arguments of the target relation. Then, the attribute-value tuples are generated by calling the corresponding predicates with argument values from the ground facts of the target relation, i.e., for each of the possible applications of the background predicates, the algorithm calls that application substituting the variables involved by the corresponding argument values of the selected example. These tuples are generalizations of the individual facts about the target relation. Then one of the possible attribute-value learning systems generates if-then rules by processing these tuples. Finally, the if-then rules are transformed into deductive hierarchical database clauses.

To allow the transformation algorithm, training examples must be ground facts that may contain structured terms, but no recursive terms. Negative examples can be generated via CWA or given by the user.

The hypothesis language in LINUS is restricted to deductive hierarchical database clauses. Certain types may appear in negated form in the body of the hypothesis clause.

Background knowledge has the form of deductive database clauses. The required variable types have to be given by the user. Background knowledge consists of two types of predicate definitions: utility functions and utility predicates. Utility functions compute a unique output value for given input, with the user declaring the input/output modes. Utility predicates are boolean functions with arguments only and for a given input they compute true or false only.

2.4.7 MFOIL

MFOIL [8] is an extension to FOIL that improves its noise handling skills integrating many noise-handling techniques adapted from attribute-value learning into FOIL, incorporating namely

the Laplace-estimate and the m-estimate to replace the information gain measure used in FOIL.

The FOIL encoding-length stopping criterion is replaced by tests of statistical significance.

MFOIL also adapts a covering strategy but it uses beam search instead of the hill-climbing search used in FOIL. Some of the most advanced techniques in FOIL are not performed by MFOIL. MFOIL supports intensionally defined background knowledge in addition to the extensionally defined supported by FOIL.

Finally, MFOIL also lets the user declare additional information regarding the background knowledge to reduce the number of candidate body literals that are constructed in induction, helping to gain efficiency.

2.4.8 PROGOL

PROGOL [22] also employs a covering approach like FOIL, selecting an example to be generalized and finds a consistent clause that covers it. All redundant clauses are removed from the theory and the example selection and generalization cycle is performed until all positive examples are covered. When constructing hypothesis, PROGOL conducts general to specific search in the θ-subsumption lattice of a single clause hypothesis. PROGOL computes the most specific clause covering the seed example and belonging to the hypothesis language (the bottom clause), and this clause bounds the lattice from below. The search strategy is basically an A*-like algorithm guided by an approximate compression measure. Each invocation returns a clause that maximally compresses the data. It can learn ranges and functions with numeric data too.

The hypothesis language is restricted by mode declarations of the user, that specify the atoms to be used in the head literals or body literals in hypothesis clauses. For each atom, the mode declaration indicates the argument types and whether it is an output variable, an input variable or a constant, bounding the number of solutions for instantiating the atom. Types are defined in background knowledge or in Prolog built-in functions.

Arbitrary Prolog programs are allowed as background knowledge. Positive examples are represented by definite clauses and negative examples and integrity constraints are represented as headless Horn clauses. PROGOL learns arbitrary integrity constraints using the CWA. It also provides a wide range of parameters to control the generalization process and allows to relax consistency with an upper bound on the number of covered negative examples.

2.4.9 TILDE

An algorithm developed at the K.U.Leuven learns a predicate logic theory using logical decision trees that are a first-order logic upgrade of the classical decision trees used by propositional learners. In the same manner as propositional rules can be extracted from decision trees, clauses can be extracted from logical decision trees. These trees can be used to classify unseen cases

directly or they can be transformed into a Prolog program.

The TILDE [4] system is a prototype implementation of this algorithm. It contains many features that are state of art decision tree techniques for attribute-value problems. With this, there are some techniques specific to ILP: a language bias specified by means of types and modes of predicates, a form of lookahead is incorporated and dynamic generation of literals is possible, that is a technique that allows, among other things, to fill in constants in a literal. It can also learn in numerical domains.

Also, TILDE is capable of performing clustering and regression, along with classification.

It supports the standard ILP setting (learning from entailment) where given a set of positive and negative examples, that are facts, and a background theory, the algorithm finds a hypothesis complete and consistent.

2.4.10 Aleph

Aleph [35] stands for "A Learning Engine for Proposing Hypothesis" and it is an ILP system that is supposed to be a prototype for exploring ideas that incorporates some functionalities of other ILP systems such as FOIL, PROGOL and TILDE.

The basic algorithm of Aleph is the following: First, it selects an example to be generalized. Then, it constructs the most specific clause that entails the example selected in the language restrictions provided, that is called the bottom clause, in the saturation step. After the bottom clause is constructed, it finds a clause more general than the bottom clause by searching for some subset of the literals in the bottom clause that has the "best" score, this is the reduction score. Finally there is a post-processing step where the clause with the best score is added to the current theory and all examples made redundant are removed.

However, the user can modify each of these steps.

The background knowledge is in the form of Prolog clauses relevant to the domain. The user can also mention language and search restrictions, such as modes, types and determinations.

The positive examples and the negative examples are ground facts.

Different search strategies and evaluation functions are supported by Aleph and not only the one mentioned in the basic algorithm explained above.

2.4.11 Comparison between the systems

In Table 2.1 we present some characteristics of each of the systems.

Name	Year	Type of Search	Saturation	Extra Intensional B.K.	Source Code
FOIL [28]	1990	Specialization	No	No	C
GOLEM [24]	1990	Generalization	Yes	No	C
LINUS [9]	1991	Propositional learning	No	Yes	Prolog
FOCL [27]	1992	Specialization	No	Yes	Lisp
Hydra [1]	1993	Specialization	No	Yes	Lisp
MFOIL [8]	1993	Specialization	No	Yes	Prolog
PROGOL [22]	1995	Specialization	Yes	Yes	Prolog
FORTE [30]	1995	Spec. / Gen.	No	Yes	Prolog
TILDE [4]	1998	Logical Decision Trees	No	Yes	Prolog
Aleph [35]	2007*	Specialization	Yes	Yes	Prolog

* 2007 is the release date of the latest version of Aleph.

Table 2.1: Comparison between the different ILP systems described in 2.4

In Table 2.1, we can see different aspects of each of the ILP systems described along this section. The first one we see is the name of the system, followed by its year of creation (that determines the order in which the systems appear in this table). Next we have the column *Type of Search*, that states whether the ILP system performs a specialization search, a generalization search, applies a propositional algorithm (only in LINUS) or uses a Logical Decision Tree algorithm, like in TILDE. We can see that in this field, many systems perform specialization searches, what is understandable, since FOIL acts that way and many of the systems present in the table are extensions of FOIL, such as FOCL and MFOIL. Other relations that are of notice is that Aleph is a successor of Progol, as stated in its manual [35], since its creation was to better understand the ideas in [22] and also the initial name of the project was P-Progol, that goes back to 1993. The next column tells us if the system uses saturation and saturated clauses in generating new hypothesis. This is important since we will be using saturation in our approach, since that to build our support data structure the way we want we need to work with bottom clauses. The only systems that use saturation are GOLEM, PROGOL and Aleph. The field *B.K.* says how each system allows the representation of background knowledge and we can see that most of them allow an intensional representation of it, which is important since it is much more concise and interesting to describe background concepts intensionally. The *Source Code* column states in which language is the system implemented. Since the authors are familiar with Prolog and C++ and there is an interface between the two languages, Prolog systems are preferred, since Lisp is an older and unfamiliar language to the authors, and the systems encoded in C are more limited, since they do not offer the same features as the more recent ones. The last column specifies the example representation of each system and, except in Progol that allows definite clauses, every system represents examples as ground facts. Other statement we have to make, that is not on the table is that when it comes to documentation and availability of source code, Aleph is the one on the lead, since its documentation [35] is very complete and organized and the source code in Prolog is very well documented.

2.5 GRAPHS AND HYPERGRAPHS

When looking for a hypothesis, the operators that construct new clauses, both generalization and refinement operators, usually do not search for them in a multidirectional manner, i.e., do not take into account all of the possible successors a given clause can have when we consider

all the variables as expansible. This is because they only add one literal at a time to a given clause and they start by putting a literal in the clause that only uses some of the head variables, instead of adding literals that relate all the head variables.

This approach works well for some problems in ILP, although it contains an arbitrary bias in which variables will be explored in more detail. However, when it comes to multi-relational problems in ILP, often all variables play an important role in describing the goal concept. With a classic way of exploring the hypothesis space, we would not be able to give all variables the same possibilities of expressing their importance in the target concept, but instead would give to a set of arbitrary variables more importance only because they appear in the first literals chosen by the algorithm, existing the possibility that some of the head variables would not even be represented in the body of the hypothesis due to previous choices of literals done with an insufficient scope. This can be solved if we look at the problem as a search problem and represent it using a graph structure. This way, one can start with multiple nodes, and, defining the relationship of neighborhood in the context of ILP clauses, one can pay attention to multiple possible expansions to the clause, which can yield in better theories than when biasing the variables chosen, given then the same opportunities to every variable to reveal their importance.

Nonetheless, there is a way to force systems that work with saturation to start with literals that relate all head variables: one can include a new predicate in the background knowledge, where every variable must appear, we create a new mode for this predicate and force the system to chose this predicate first (for example, in Aleph, all it takes is to place it at the beginning of the background knowledge). However this is not a very aesthetic solution and, although we are pushing the system into relating all variables in the head of the target relation, we still have no way of making sure that the algorithm will pick some other literals that will, eventually, relate every head variable and there is a risk that some of these variables only appear on this fake predicate included to convince the algorithm to search for literals that relate all the head variables and that, ultimately, does not have any meaning whatsoever. In order to provide a more elegant solution to this problem, we will resource to hyperpath finding [26]. Next sections introduce some basic concepts on graphs and hypergraphs necessary to understanding what follows next.

2.5.1 Graphs

A graph, from which Figure 2.8 is an example, is a set of points and a set of arcs with each arc joining two points. We will call the points vertices and the arrows arcs.

The set of vertices will be denoted by V and the set of arcs by A.

An arc that connects vertices a and b is denoted by (a, b).

So we formally define a graph $G(V, A)$ as:

- V is a set $\{v_1, v_2, \ldots, v_n\}$ of vertices and

- A is a set $\{(v_i, v_j), (v_m, v_n), \dots\}$ of arcs that are elements of the Cartesian product $V \times V$ where (x, y) means there is an arc connecting the vertices x and y.

The number of vertices in a graph is called the order of the graph.

An arc of G of the form (x, x) is a loop.

In a directed graph G with an arc (x, y) we call x the initial endpoint and y the terminal endpoint.

A vertex y is a successor of a vertex x if there is an arc of the form (x, y), where x is an initial endpoint and y a terminal endpoint.

Similarly, y is a predecessor of x if there is an arc of the form (y, x).

The set of neighbors of x is the union of the set of the successors and predecessors of x.

If the directions are not specified, then we call it a multigraph or undirected graph and we deal with the pair (X, E), where E is a set of edges.

Two edges (or arcs) are called adjacent if they share at least one endpoint.

If a vertex x is the initial endpoint of an arc u, that is not a loop, we say the arc u is incident out of x.

The degree of a vertex x is the number of arcs with x as an endpoint.

A chain is a sequence of arcs of G such that each arc in the sequence has one endpoint in common with its predecessor in the sequence and its other endpoint in common with its successor. The length of the chain is the number of arcs in the sequence.

A path of length q is a chain in which the terminal point of an arc u_i is the initial endpoint of the arc u_{i+1} for all $i < q$.

A cycle is a chain such that no arc appears twice in the sequence and the two endpoints of the chain are the same vertex.

A connected graph is a graph that contains a chain for each pair x, y of distinct vertices.

A vertex s can reach a vertex t (and t is reachable from s) if there exists a sequence of adjacent vertices (i.e. a path) which starts with s and ends with t.

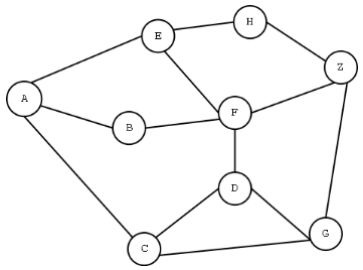


Figure 2.8: Example of a graph

This is the only theory about graphs one must be familiar with to understand our approach.

2.5.2 Hypergraphs

In this section we will state some definitions of hypergraphs that will be useful in understanding some parts of our work.

We will start by defining a directed hypergraph.

Definition 42. A directed hypergraph $\mathcal{H}$ is a pair $\langle N, H \rangle$ where N is a set of nodes and H a set of hyperarcs. A hyperarc is an ordered pair $\langle S, t \rangle$ from an arbitrary nonempty set $S \subseteq N$ that is called the *source set* to a single node $t \in N$ that is called the *target node*. $\Diamond$

A directed graph is a special case of a directed hypergraph where all source sets have cardinality one.

We now list some parameters to be taken into account for directed hypergraphs:

- $n = |N|$ is the number of nodes.
- $h = |H|$ is the number of hyperarcs.
- $a = \sum_{S \in \mathcal{S}} |S|$ where $\mathcal{S}$ denotes the set of source sets: $\mathcal{S} = \{S | \langle S, t \rangle \in H \text{ for some } t \in N\}$ is the sum of the cardinalities of all the source sets, called the *source area* a.
- $a' = \sum_{S \in \mathcal{S}_M} |S|$ where $\mathcal{S}_M$ denotes the set of nonsingleton source sets: $\mathcal{S}_M = \{S | \langle S, t \rangle \in H$ for some $t \in N$ and $|S| > 1\}$ is the *nonsingleton area* a'.
- the size s is the overall length of the description of the hypergraph, denoted also as $\mid \mathcal{H} \mid$. If the hypergraph is represented by adjacency lists we have that $\mid \mathcal{H} \mid \equiv s = n + a' + h$.

We will now define a subhypergraph.

Definition 43. Let $\mathcal{H} = \langle N, H \rangle$ be a directed hypergraph. A hypergraph $\mathcal{H}' = \langle N', H' \rangle$ such that

- $N' \subseteq N$
- $H' \subseteq H$ and, for each $\langle S, t\rangle \in H', S \subseteq N'$.

is called a *subhypergraph* of $\mathcal{H}$. We denote this by $\mathcal{H}' \subseteq \mathcal{H}$. ◊

Definition 44. Let $\mathcal{H} = \langle N, H\rangle$ be a directed hypergraph and let $H' \subseteq H$ be a set of hyperarcs in $\mathcal{H}$. Let $N' \subseteq N$ be the union of source sets and target nodes of hyperarcs in H'. The hypergraph $\mathcal{H}' = \langle N', H'\rangle$ is said to be the *subhypergraph* of $\mathcal{H}$ induced by $\mathcal{H}'$. ◊

We have an example of a hypergraph and subhypergraph in Figure 2.9.

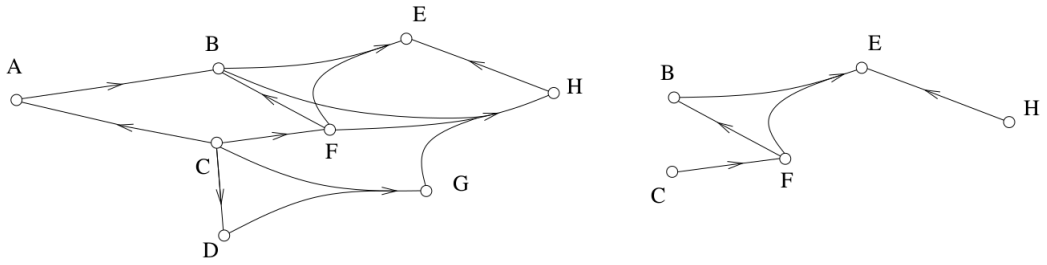


Figure 2.9: Example of a hypergraph on the left and a subhypergraph on the right. Image taken from [3].

We will now define a hyperpath in directed hypergraphs, differentiating between *folded* and *unfolded* hyperpaths.

Definition 45. Let $\mathcal{H} = \langle N, H\rangle$ be a directed hypergraph, $X \subseteq N$ be a non-empty subset of nodes and y a node in N. There is a *hyperpath* from X to y in $\mathcal{H}$ if

- either $y \in X$ (extended reflexivity).
- or there is a hyperarc $\langle Z, y\rangle \in H$ and hyperpaths from X to each node $z_i \in Z$ (extended transitivity).

◊

We now describe a *hyperpath tree* that is a tree labeled in the nodes, that naturally describes the above definition of hyperpath.

Definition 46. Let $\mathcal{H} = \langle N, H\rangle$ be a directed hypergraph, $X \subseteq N$ be a non-empty subset of nodes and y be a node in N such that there is a hyperpath from X to y. A hyperpath tree from X to y is a tree $t_{X,y}$ defined as:

- if $y \in X$ then $t_{X,y}$ is empty.

- if there is a hyperarc $\langle Z, y\rangle \in H$ and hyperpaths from X to each node $z_i \in Z$, then $t_{X,y}$ consists of a root labeled with hyperarc $\langle Z, y\rangle$ having as subtrees the hyperpath trees t_{X,z_i} for each node $z_i \in Z$.

◊

The hyperpath tree $t_{X,y}$ is such that its root has the target node y in the label. Also, if $\langle S, t\rangle$ is in the label of a leaf in the hyperpath tree, the source set S is contained in X. The hyperpath tree is the unfolded representation of a hyperpath, since it explicitly describes the sequence of hyperarcs as traversed while going from X to y and the same hyperarc may appear many times in the hyperpath tree.

We now present a more concise description for hyperpaths.

Definition 47. Let $\mathcal{H} = \langle N, H\rangle$ be a directed hypergraph. Let $X \subseteq N$ be a non-empty subset of nodes of $\mathcal{H}$ and $y \in N$ be a node such that there is an hyperpath from X to y in $\mathcal{H}$. A folded hyperpath $h_{X,y}$ from X to y is given by the subhypergraph of H induced by the hyperarcs in the unfolded hyperpath $t_{X,y}$. ◊

Based on definition 47 we can say that either $h_{X,y}$ is the empty hypergraph or there is an hyperarc $\langle Z, y\rangle$ in $h_{X,y}$ and, for each node $z_i \in Z$ there exists an hyperpath h_{X,z_i} from X to z_i which is a subhypergraph of $h_{X,y}$.

Below we show an unfolded and a folded hyperpath present in the hypergraph given in Figure 2.9, shown in Figure 2.10.

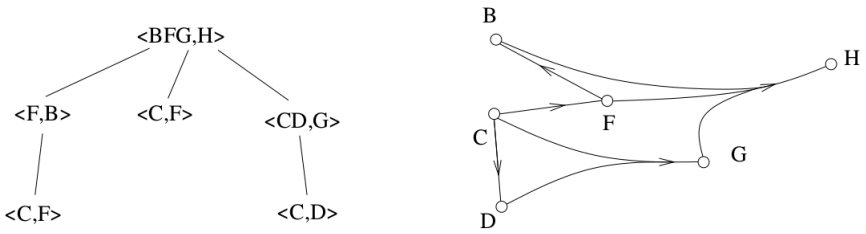


Figure 2.10: Unfolded (left) and folded (right) hyperpath from C to H. Image taken from [3].

We now define the *size* of a hyperpath, that gives the overall length of the description of $h_{X,y}$.

Definition 48. The size of a folded hyperpath $h_{X,y} = \langle N_h, H_h\rangle$ is the sum of the number of hyperarcs and source area, so $s(h_{X,y}) = |H_h| + \sum_{S_i \in \mathcal{S}()h_{X,y}} |S_i|$ where $\mathcal{S}(h_{X,y})$ is the set of all source sets in $h_{X,y}$ ◊

In general we may have several hyperpaths from a source set X to a single node y. If we have a measure function μ, a natural problem consists of selecting the μ-*optimal* hyperpath according to an optimization criterion (max or min).

We can now define a notion of distance among the nodes in the hypergraph, given a measure function μ for hyperpaths.

Definition 49. Given a source set $X \subseteq N$ and a target node $y \in N$ of a directed hypergraph $\mathcal{H}$ we define $\delta_{\mathcal{H}}(X, y) = min_{h^i_{X,y}}\{\mu(h^i_{X,y})\}$ where $h^i_{X,y}$ ranges over any possible hyperpath between X and y in H and $\delta_{\mathcal{H}}(X, y)$ is ∞ if there is no such hyperpath. $\Diamond$

Chapter 3

State of the Art

In this chapter we will talk about some of the work already done by others that served as a basis in many phases of our work.

3.1 RELATED ILP SEARCH ALGORITHMS

In this section we will talk about the relational Pathfinding and an extension to it that it is called mode directed pathfinding.

3.1.1 Relational Pathfinding

One of the biggest problems in top-down inductive learners is that they rely in heuristics based on hill climbing, which makes them vulnerable to local plateaus and local maxima. Relational pathfinding [29] is designed so ILP systems do not get stuck in local maxima when these local maxima does not represent a contribution to the learning process.

Remember that in top-down systems each clause is generated by adding one literal at a time using a hill-climbing-based technique and at each step, instantiations of each predicate in the data are tested for their ability in discriminating the remaining positive and negative examples and the one that performs the best is added to the clause. This hill-climbing approach is what makes these approaches vulnerable to local maxima and plateaus.

The idea of pathfinding, whose algorithm is in Algorithm 2 is to view a relational domain as a graph of constants linked by the relations that hold between the constants.

It bases on the idea that important concepts will be represented by a small number of fixed paths among the constants defining a positive instance.

Algorithm 2 Overview of the relational pathfinding algorithm

```
1: instantiate a rule with a positive instance
2: find isolated sub-graphs
3: for all sub-graph do
4:     constants become initial end-values
5: end for
6: repeat
7:     for all sub-graph do
8:         expand paths by one relation in all possible ways
9:         remove paths with previously seen end-values
10:    end for
11: until intersection is found or resource bound is exceeded
12: if intersection exists then
13:    for all intersection do
14:        add path-relations to original rule
15:        if new rule contains singletons then
16:            add relations that use the singletons
17:            if all singletons are eliminated then
18:                keep the expanded rule
19:            else
20:                discard the rule
21:            end if
22:        end if
23:        replace constants with variables
24:    end for
25:    select most accurate rule
26: end if
27: if negatives are still provable then
28:    use normal specialization to finish the rule
29: end if
```

We will now explain Algorithm 2 in a few paragraphs.

This algorithm finds paths by expansion around the nodes associated with the constants in a positive example. It chooses a positive instance and uses it to instantiate (line 1) the initial rule, then if identifies isolated sub-graphs among these constants (line 2) and if the initial rule does not contain antecedents, each constant forms a singular sub-graph (lines 3-5).

Each sub-graph is viewed as a nexus from which the surrounding portion of the domain graph is explored. Each exploration that leads to a new node in the domain graph is a path and the value of the node it has reached is the end value of the path. Initially, constant in a sub-graph is the end-value of a path with zero length.

Taking each sub-graph in turn, all new constants that can be reached by extending a path with a defined relation are found and these new constants form the new set of path end-values for the sub-graph (lines 7-10). Then the algorithm checks this set of end-values against the sets of end-values of all other sub-graphs, looking for an intersection. If there is no intersection, the next node is expanded. This algorithm loops until an intersection is found or a resource bound is exceeded (lines 6-11).

When an intersection is found, the relations in the intersecting paths are added to the original instantiated rule (line 14). If these relations introduce new constants that appear only once (singletons), the rule is completed by adding relations that hold between the singletons and other constants in the rule (line 15-16). If the singletons can not be all used, the rule is rejected (lines 19-20). Finally, the algorithm replaces all constants with variables (line 23) to produce the final theory clause. If several intersections are found simultaneously, clauses for each of those intersections are generated and the one with the best accuracy in the training set is chosen (line 25).

In theory this algorithm is exponential in the number of nodes in the sub-graphs, since it expands in every way at each turn, but in practice the authors defend that generally it is successful for two reasons: since paths are expanded by one relation in all possible ways, the total number of paths to be explored before an intersection is encountered is reduced and the second is that most important relations are defined by short paths, limiting the depth of the search.

By running FORTE with and without relational pathfinding, the authors came to conclude that using relational pathfinding translates in a significant gain in learning performance for any size of the training set for the family dataset used in [28], improving the accuracy of the theory significantly, since the accuracy, without pathfinding, would go above the 85%, not even with 300 instances, and with pathfinding it got to 100% with only a few hundreds of instances.

With the relational pathfinding algorithm, we get a model that gives us the intuition necessary to build our work on: that good definitions of concepts generally are not very deep, i.e., do not require a deep search when expanding clauses and looking for new theories, and also relate the target variables, because behind that relation of the target variables, generally, there is knowledge. This is the main principle in our system, explained in more detail in Chapter 4.

3.1.2 Mode directed pathfinding

In mode directed pathfinding [26], the algorithm does not act directly over the background knowledge but over the saturated clause of a given example. This saturated clause is obtained using mode declarations, that, in a nutshell, mean that a literal can only be added to a clause if the input variables of the literal are bounded, embedding directionality in the graph formed by literals. The authors say that a saturated clause can be described as a direct hypergraph and show that path finding reduces to reachability in the hypergraph, where each hyperpath corresponds to a hypothesis and propose an algorithm to enumerate all such hyperpaths.

The authors start by mapping a clause into a hypergraph in the following manner:

- Each node corresponds to a literal L_i.
- Each hyperarc with $N' \subseteq N$ nodes is generated by a set $\mathcal{V}$ of $i-1$ variables $V_1, \ldots, V_{i-1}$ that appear in literals $L_1, \ldots, L_{i-1}$. The mapping is such that every variable $V_k \in I_i$ appears as an output variable of node L_k, $k < i$ and I_i represents the set of input variables of the literal L_i.

This definition says that nodes in $L_1, \ldots, L_{N'-1}$ with output variables that generate input variables for the node $L_{N'}$ will be connected by hyperarcs.

We will explain this better with an example from the paper:

Before the algorithm is applied, a transformation procedure is executed in the hypergraph, as shown in Figure 3.1. Basically, what happens is that a mapping as described before is performed in order to transform the bottom clause into a hypergraph. Then, according to the modes of the target relation (the root of the graph in the left graph of Figure 3.1), the root node is divided in as many nodes as input variables the target relation contains. With this division, each new root node will now be connected to the nodes the previous unique root node was connected that depended on the variable it now represents to respect the conditions of the mapping. Figure 3.1 represents an example taken from [26].

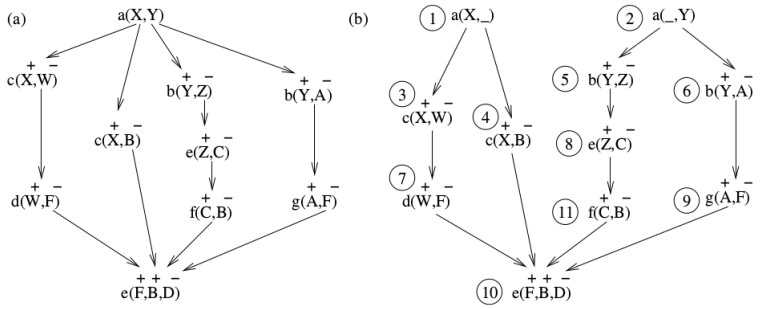


Figure 3.1: Transformation (on the right) of the hypergraph presented on the left. Image taken from [26]

.

In Figure 3.2 there is a graphical explanation of the algorithm applied to the transformed hypergraph in Figure 3.1, provided by the authors of the paper.

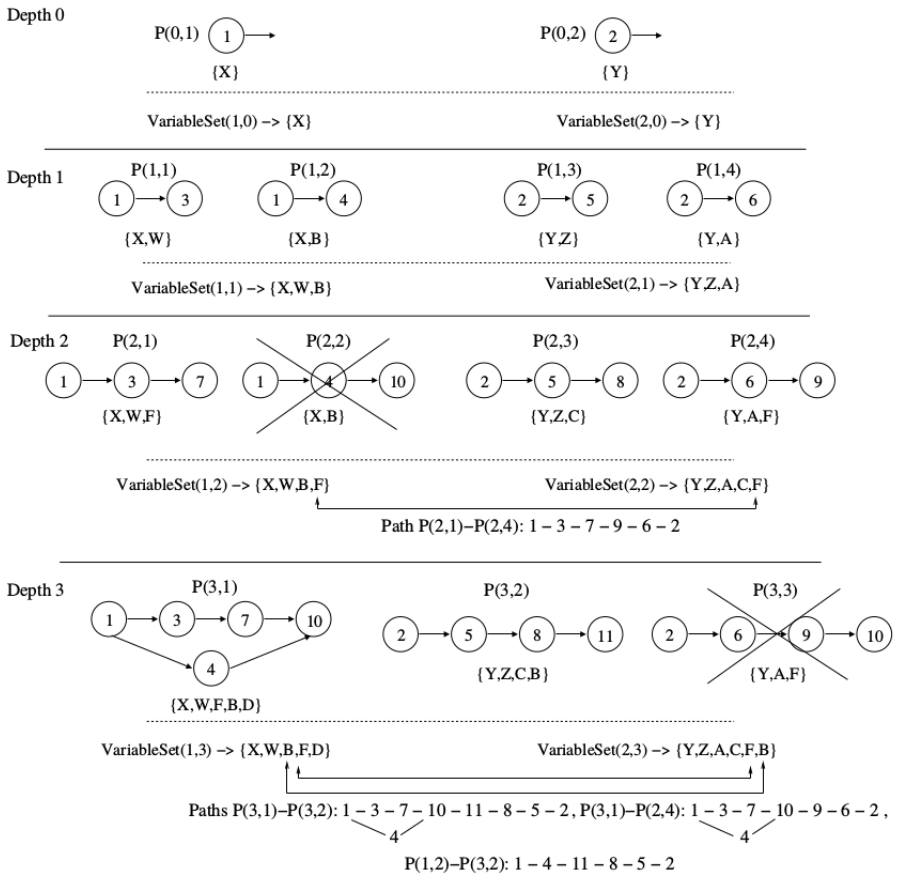


Figure 3.2: Illustration of the mode pathfinding algorithm. Image taken from [26]

The numbered nodes in the Figure 3.2 correspond to the labels of literals in the hypergraph.

The depth of the graph is indicated on the left side. Current paths are expanded at each depth if the node to be expanded has its input variables bound, otherwise they are crossed out (for example in $P(2,2)$ and $P(3,3)$). $VariableSet(s,d)$ represents the set of variables reachable from s at depth d and is used to find common variables between variable sets of different sources at a given depth. Hyperpaths found so far are indicated by $P(d,n)$, where d is the depth and n the node index at depth d. Paths found are denoted by $P(d,n_1) - P(d,n_2)$.

For each source and each depth, starting at depth 1, the algorithm does:

- Expand paths from previous *depth* $(d-1)$ if input variables for the newly expanded node, n, exist in $VariableSet(s,d-1)$ of the previous *depth* and are bound by parents reachable at the current depth.
- Place all variables reachable from s at current depth d in $VariableSet(s,d)$.
- Check $VariableSet$ of each source at the current depth for an intersection of variables and for each variable in the intersection create paths from permutations of all nodes containing

the variable, including those from previous depths.

In depth 0 there is the initial configuration with 2 source nodes, 1 and 2, represented by $P(0,1)$ and $P(0,2)$. The variables of those sources are put in their respective $VariableSets$.

The algorithm begins at depth 1, where node 1 has two hyperpaths of size 2: one to node 3 ($P(1,1)$) and another to node 4 ($P(1,2)$). Node 2 reaches nodes 5 and 6, yielding $P(1,3)$ and $P(1,4)$. At this depth it is possible to reach W and B from X, and Z and A from Y, indicated by $VariableSet(1,1)$ and $VariableSet(2,1)$. Since there is no intersection, no paths are built.

At Depth 2 the nodes 3,4,5 and 6 are expanded and since $P(1,1)$ reaches node 7, X reaches variable F. $P(1,2)$ tries to expand to node 10, but node 10 requires F and B as input and $P(1,2)$ only has B, so $P(2,2)$ is crossed out. $P(1,3)$ can be expanded with 8 and $P(1,4)$ with 9. Here there are hyperpaths from the first argument reaching W, B and F and from the second argument reaching Z, A, C and F. Since it is possible to get to F from X and Y, there is a path and if $P(2,1)$ and $P(2,4)$ are combined, it yields the first path: 1,3,7,9,6,2 ($P(d,n_1) - P(d,n_2)$)

In Depth 3, $P(2,1)$ can reach node 10 if merged with $P(1,2)$, creating $P(3,1)$ that reaches X, W, F, B and D. $P(2,3)$ is expanded to include node 11, yielding $P(3,2)$, but $P(2,4)$ cannot be expanded to include node 10, since it does not contain B, so $P(3,3)$ is crossed out. $P(3,1)$ reaches X, W, F and B and $P(3,2)$ also contains B, creating a new path. Also, $P(3,2)$ can be combined with $P(2,4)$ since they share F. $P(3,2)$ and $P(1,2)$ share B, so $P(3,2) - P(1,2)$ can be generated as well. If a hyperpath is already a path, it can be extended by merging it with another hypergraph, yielding non-minimal paths.

When it comes to experimental results, mode directed pathfinding finds longer clauses not considered by Aleph and both systems find different best clauses using the f-score measure. When it comes to test set performance, since the clauses generated by pathfinding seem to be longer, they are more vulnerable to overfitting, however recall values are generally high, but precision values are not so good, in general.

With Mode Directed Pathfinding, we got the idea of applying the Pathfinding algorithm to a saturated clause, instead of working with examples directly. This allows for the possibility of mapping each examples, or better saying, the bottom clause of each example, into a hypergraph that represents the example in a more general way and a much more structured one, emphasizing the relations between the literals according to their mode definitions and variables. Also, many problems, such as overfitting and keeping the search divided, originating longer clauses when connecting the head variables, were surpassed in our approach.

3.1.3 HOC algorithm

In [32] we have two algorithms: the first to compute all the sets of support clauses for the head output variables until depth D (Algorithm 3), that performs basically a breadth-first search over the hypothesis space defined by the bottom clause, traversing this space through the successor

subclauses of the current subclauses to visit, being the initial clause to visit the head clause. In this algorithm no coverage computation is taken during the search.

Algorithm 3 Procedure to compute the support clauses of C until a maximum depth D

Compute sets of support clauses Input: Clause C and depth D

```
ClausesToVisit = C's head
SuppClauses = {}
Depth = 1
while ClausesToVisit ≠ {} and Depth<D do
    ClausesToVisitNext = {}
    for all c ∈ ClausesToVisit do
        SuccClauses = Successors of c with respect to C
        CurSuppClauses = HOC clauses ∈ SuccClauses
        ClausesToVisitNext = ClausesToVisitNext ∪ SuccClauses - CurSuppClauses
    end for
    ClausesToVisit = ClausesToVisitNext
    Depth = Depth + 1
end while
```

Output: SuppClauses

Next, we present, in Algorithm 4, the main loop of the HOC special purpose ILP system.

Algorithm 4 Main loop of HOC ILP system algorithm

HOC ILP system algorithm Input: Positive examples E^+, mode declarations M and background knowledge B

```
Hyps = {}
for all e ∈ E+ do
    C = ⊥e
    Hype = support set of clauses for C until depth = maximum clause length
    Hyps = Hyps ∪Hype
end for
Theory = Greedy search on Hyps for the highest scoring clauses
```

Output: Theory

Since both these algorithms are pretty clear, due to the way they are written, there is no need to detail to much the explanations. However, a simple explanation for each of them will be provided.

Algorithm 3 receives as input a clause C and a value of maximum depth D. Then, in the while loop, that iterates through all the potential parts of future support clauses, all successors of each of those parts are generated through BFS. Then we see which of them are now support clauses, by determining which are HOC clauses and update the clauses to visit next as the ones

that are not, yet, support clauses, since they are potential parts of future support clauses. The while loop terminates when there are no more candidates to support clauses left to process or when the value of depth D is achieved.

In Algorithm 4 the set of hypothesis is set to empty and then for every example in the set of positive examples we obtain a clause by saturating that example, find the support clauses for that bottom clause using Algorithm 3 and add the resulting support clauses to our set of hypotheses. In the end, the set of candidate hypotheses is searched using a greedy search and the one with the highest score is returned.

In generating theories the greedy search performed on Hyps is to maximize a clause evaluation function, that, by default, is compression. Note that this algorithm aims at, essentially, inducing numerical functions, since they are one of the most prevailing classes in HOC problems.

In terms of results, this approach was tested in inducing Fibonnaci and Binomial Coefficient predicates, and it took 0.4 seconds to learn Fibonnaci and 2.1 seconds to learn Binomial, while Progol could not learn none of the concepts even after spending several hours trying and Aleph crashed in both examples after a few seconds. The predicates that were induced by HOC were the expected ones for both problems.

By studying HOC problems, we got a better notion of how to construct and apply our metric, in the sense that HOC clauses search for chains of literals linked by their variables, according to mode definitions. This is partially what we aimed to do, but not only for HOC problems, i.e., not necessarily with an output head variable present in the target relation. The metric used in our problems constructs chains of literals in a very similar way to the HOC algorithm, but uses a hypergraph as a support structure to make operations like accessing the children of a given literal much more simple.

Chapter 4

Design and Implementation

The only ILP approaches found in the literature that focus on learning multi-variable concepts are the three mentioned in the previous chapter. HOC is more focused on allowing to learn numerical functions. The two works most related to ours are Relational Path Finding and Moded Path Finding.

In this chapter we describe our own implementation of an ILP system that can handle multi-variable concepts. Our design and implementation follow the main generic ILP algorithm described in Section 2.2.7 and adopts the best practices developed by Ong et al [26]. However, our implementation differs from the implementation in [26] essentially when searching the space of possible clauses, in the sense that our work searches it multidirectionally, allowing for simpler theories while the other algorithm searches it unidirectionally, as we will explain in more detail in the rest of the chapter.

4.1 OUR ALGORITHM

We will begin by presenting our algorithm for searching multi-variable concepts given a set of positive examples, a set of negative examples, mode declarations and background knowledge. Such algorithm is presented in Algorithm 5.

Algorithm 5 Skeleton of the algorithm developed in this work

Compute the set of clauses that maximize the metric given in 4.3.2

Input: Mode definitions M, set of positive examples E, set of negative examples N, background knowledge B

for all $e \in E$ **do**
  $C = saturate(e)$
  $Hypergraph = construct\_hypergraph(C, M)$
  $BestClauses = search\_hypergraph(Hypergraph)$
end for

Output: BestClauses

Although this algorithm seems really short, there are some things that were already said in the previous chapters about each of these steps. Each of the steps of the algorithm will be slightly explained in the next paragraphs, having a much more thorough, line by line, explanation in the next section.

The procedure $saturate(e)$ is, actually, the procedure that given an example e returns the bottom clause, i.e., the most specific clause that covers the example e given as argument. This procedure is actually implemented by Aleph [35] and it consists of an implementation of the theory previously presented in 2.2.14.

The $construct\_hypergraph(C, M)$ constructs a hypergraph following the algorithm and properties described in 3.1.2. For completeness purposes, we will, again, briefly describe such construction, in an algorithmic format in Algorithm 6.

Algorithm 6 Constructs a hypergraph that relates the literals in the bottom clause according to their variables and respective mode definitions

consctruct_hypergraph

Input: A clause C and a set of mode definitions M

1: $Nodes = \emptyset$
2: $Edges = \emptyset$
3: **for all** literal $l \in C$ **do**
4: $\quad Nodes = Nodes \cup Make\_Node(l)$
5: **end for**
6: Create an edge that connects the root node, i.e., the node representing the head literal, to every node that contains, as input variables, the input variables of the previously mentioned root node and add it to $Edges$
7: **for all** node $n \in Nodes$ **do**
8: $\quad Out_n = Out(n)$ ▷ Out_n now contains the output variables of the literal represented by the node n
9: $\quad$ **for all** variable $v_o \in Out_n$ **do**
10: $\quad\quad$ Create an edge that connects n to a node m that contains v_o as an input variable and add it to $Edges$
11: $\quad$ **end for**
12: **end for**
13: $Hypergraph = (Nodes, E)$

Output: $Hypergraph$

Notice that the hypergraph structure described and returned in Algorithm 6 will be composed by a set of Nodes and a set of Edges, i.e., it will be a graph. However, the conditions required to check if a given literal has its input variables all instantiated in a given clause (checking if the clause is IO consistent 2.2.15), which is a core part when generating future specializations to the clause, will force different literals (source nodes) that contains as output variables the input variables of a given literal to be connected to it. In a nutshell, we create a graph describing

all the connections between the literals, but since each literal has variable dependencies from previous literals in the bottom clause, this graph will behave, in practice, as a hypergraph.

The most relevant contribution of our algorithm goes with the *search_hypergraph* procedure, where we search the previously constructed hypergraph in the manner described in Algorithm 7.

Algorithm 7 Searches the hypergraph to find the set of best clauses that it can contain *search_hypergraph* **Input:** Hypergraph H

1: $Seeds = generate\_seeds(H)$ ▷ Obtains sub-clauses of the bottom clause such that those sub-clauses contain all the head variables
2: $maxScore = 0$
3: $BestClauses = \emptyset$
4: **for all** $s \in Seeds$ **do**
5: $spec_s = specialize(s)$
6: **if** $score(spec_s) > maxScore$ **then**
7: $maxScore = score(spec_s)$
8: $BestClauses = \{spec_s\}$
9: **end if**
10: **if** $score(spec_s) == maxScore$ **then**
11: $BestClauses = BestClauses \cup \{spec_s\}$
12: **end if**
13: **end for**

Output: BestClauses

In the end of this procedure, a set of clauses that have the higher score, according to a given metric, is returned. Since this is a novel algorithm, here we will just present it, and will elaborate our explanation on its behavior in the next section.

Just by leaning a little bit on the presentation of the algorithm, one easily notices two things: it differs from the one in [26] and from the generic algorithm in Section 2.2.7 and also it has a really high potential for finding good multi-variable concepts, in the sense that the searching algorithm seems to really benefit those types of clauses (the metric also helps a lot, but we will leave that for further on the document).

From the Mode Directed Pathfinding algorithm in [26], the main difference is the way we perform the search of the hypergraph. The seed generation process, which we will explore in more detail later, is really a game changer, in the sense that it generates initial candidate clauses, that are sub-clauses of the bottom clause, that already contain all the head variables, possibly not connected yet, allowing us to perform a multidirectional search, considering every possible child or parent of each literal in the clause, while Mode Directed Pathfinding searches for them unidirectionally in the sense that only the last of the literals added to the clause will be expanded for possible new literals to add to the clause, as well as other clauses previously created.

This change in the search is intended to search the space in a wider way, since every parent

and child of every literal can be added at a given moment if it enhances the hypothesis. Also, presenting such way will lead to, potentially, smaller clauses than Mode Directed Pathfinding, a problem stated by the authors that could, easily, lead to overfitting, since we add one literal at a time, while Mode Directed Pathfinding concatenates already formed hyperpaths, which yields much longer clauses than adding only one literal at a time to the candidate clause and evaluating it step by step, i.e., before adding a new literal.

When comparing to the generic algorithm in Section 2.2.7, there are many core differences. First, it begins with the empty clause as initial theory, while we generate clauses that contain all head variables in their literals. Also, we begin by saturating every example, which is not done in the generic algorithm. Another major change is the fact that the generic algorithm performs cover removal, i.e., removes all positive examples covered by a temporary theory, while our approach does not. Although it is pretty standard to do so, it is pretty clear that just because a positive example is covered by a given theory obtained through the processing of another positive example, does not mean that it can not generate another different, possibly better, theory when submitted to our algorithm. For that reason we decided not to add cover removal to our core algorithm, although we implemented and tested this setting in our algorithm, as we can see in the next chapter. However, using cover removal, although it does not guarantee completeness in the sense that every possible theory was generated by our algorithm for a given set of positive examples, can really make the algorithm go faster, so we also implemented a version that implements cover removal, which has proven to be of high value for a lot of cases. Nonetheless, the results of its use will be determined by the quality and consistency of the dataset, in the sense that high levels of noise can lead the algorithm to prematurely exclude important clauses to generate a quality theory due to the presence of a high number of noisy clauses. Another solution explored by us was running the algorithm with only a subset of the set of positive examples. More details on these results are in Chapter 5.

4.2 A STEP BY STEP EXPLANATION OF OUR ALGORITHM

In this section we will perform a much more detailed explanation of the algorithm step by step, recalling some of the theoretical aspects available in previous sections.

As seen in 4.1, we have that the main algorithm 5 consists of, essentially, three parts: saturation of a positive example, hypergraph construction and hypergraph search. When it comes to saturation, since it is a very theoretical aspect already explained in detail in previous sections 2.2.14, we assume the reader already is familiar with the saturation process and what it represents, so we will not, once again, present with a detailed theoretical explanation but, instead, we will state only that saturating an example corresponds to generating the most specific clause that covers that example, i.e., corresponds to generalize minimally the example.

However, a practical example along with an example dataset of the saturation process is provided in Example 24.

Example 24. Given the following background knowledge, three positive examples (cases that are known to be true), three negative examples (cases that are known to be false) and mode declarations:

Positives	**Negatives**	**Background Knowledge**	**Modes**
gp(bob,peter)	gp(peter,alice)	p(bob,tom)	modeh(*,gp(+p,+p))
gp(alex,alice)	gp(megan,alex)	p(tom,peter)	modeb(*,p(+p,-p))
gp(megan,richard)	gp(tom,bob)	p(alex,olivia)	determination(gp/2,p/2)
		p(olivia,carl)	
		p(olivia,alice)	
		p(megan,sarah)	
		p(megan,jennifer)	
		p(jennifer,richard)	
		p(jason,michael)	
		p(richard, amanda)	
		p(richard, lisa)	
		p(bob,marge)	

We can then produce the following bottom clause for first example *grandparent(bob,peter)*:

gp(A,B) :- p(A,C), p(A,D), p(C,B).

◀

So we will essentially approach in high detail the other two main parts of the algorithm: the hypergraph construction and the hypergraph search.

4.2.1 Hypergraph construction

We will present an analysis to the hypergraph construction algorithm used in our system. Such algorithm is Algorithm 6, where the *construct_hypergraph* procedure is displayed.

In order to create the hypergraph that relates literals in the bottom clause according to their variables and their mode definitions, we need as input a clause c, that is, in this case, a bottom clause representing one given positive example, and the mode definitions for each of the predicates that are present in the bottom clause, so we know if a given variable is an input variable or an output variable.

The first condition such hypergraph must follow is that every literal is a node in the hypergraph. That is accomplished in the lines 3-5 of the algorithm, where we add to the initially empty set of nodes of the hypergraph (named $Nodes$ in our algorithm) a new node for each literal in the bottom clause that represents such literal. Consider the $Make\_Node(l)$ procedure such that given a literal it returns a singleton Node representing the literal l.

In line 6 we create the first two levels of our graph, where each node containing as input variables a subset of the head literal input variables is connected to it. The reason why this is made this is way is as follows. Since this hypergraph is supposed to represent the bottom clause in such way that every hyperpath on it represents a succession of literals that, together, form a clause whose variables are all instantiated, it makes sense to make the root node the head literal, since it contains the target goal. The only connections made to other literals, in the case of the root node are slightly different than those made between other nodes. That happens for essentially two reasons: first there not many cases where we want to learn a concept whose goal predicate contains output variables, being that class of problems a very restricted one, as stated in 2.2.15; also, following the logic of mode declarations, we have that when attributing values to the head variables, we can only attribute them to the ones that are input ones, so we generate the output values for such input values, if there are any output variables in the goal predicate, meaning that it makes no sense to assume a connection of the head literal to other literals based on its output variables, for it may create clauses that may not be possible to instantiate, due to the presence of unbound variables, i.e., variables that can take any value and in literals that can break, in a way, the chain of literals that this hypergraph aims to construct. Also, when interpreting the meaning of having an output variable in the head literal, we easily conclude that it means that the goal literal will receive a set of input variables and return a set of output variables, i.e., it will produce such output variables through the literals the concept relates. Since those variables will be produced, it makes no sense analyzing a chain of literals when the variables connecting those literals are output variables of the goal target and, therefore, will be produced by literals containing the input variables in the target concept instead of producing such variables. All such connections are represented as edges on a graph relating two nodes and added to the initially empty set of edges E. More information on why the first two levels are built this way can be found in [26].

In lines 7-12, we create the other relations between the literals, where such relations consist in connecting nodes that share variables and in one of those nodes it is an input variables and in the other it is an output variable, i.e., connecting two nodes that share a variable that has two different mode definitions. This creates a chain of literals whose relation holding the chain is the one of generating new literals due to the production of new variables. In other words, these chains of literals will tell us whose literal produces variables that are essential to other literals so the variables are bound when appearing in a future clause, since they are input variables in them. To do so, we search through all the nodes and for each node n we access its output variables through the procedure $Out(n)$ (line 8) where given a node it returns the output variables of the term that n represents. After we have all the output variables, we create a connection from n to any node m that contains any of its output variables as an input variable and add that connection to the set of edges E.

Finally, in line 13, we return the hypergraph represented by the set of nodes and the set of edges. Such hypergraph is really a graph, but the multiple dependencies that arise so that every literal has every input variable previously produced by another literal or simply given from the outside through the input variables of the goal target, really transform this directed graph into

an hypergraph, since, in order to reach a certain node, one requires to reach first a set of other nodes (source nodes) containing all the input variables of the literal represented by the target node, and this is a property only achievable when considering a hypergraph.

The final structure will follow the principles in the hypergraph construction displayed in [26] and 3.1.2.

An example of the hypergraph construction adopted in our system is available in Figure 4.1.

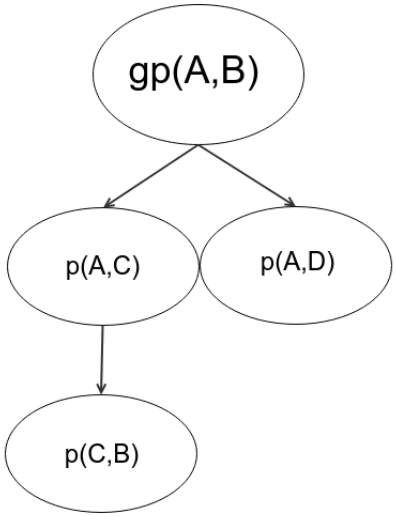


Figure 4.1: Hypergraph construction for the bottom clause presented in Example 24.

4.2.2 Hypergraph search

As in 4.2.1, we will start by presenting, again, the algorithm for the search in the hypergraph nodes, this time with numbered lines. Such algorithm is Algorithm 7.

In order to return a set of clauses that maximize a given heuristic, our algorithm uses only the hypergraph created with the *construct_hypergraph* procedure.

In line 1, we generate a set of seeds from the information in the Hypergraph H. These seeds are all possible sets of literals so that every input variable in the goal concept is present in such set. Since our goal is to create a system that relates all head variables through literals in which they appear and the chains of literals representing the connection through the variables of those literals, our starting point in discovering such concepts is a clause that already has, partially, what we aim for, in the sense that it contains all head variables. The next efforts of the algorithm, since every candidate clause already has all interest variables in it, will be in relating all those variables, since they, probably, are not when the seeds are generated.

In the cycle in lines 4-13, we have the process of specializing multiple times each of the seeds,

in order to achieve two goals: a higher score in a given metric and a relation between all the head variables.

Since this is the main cycle and the main novelty in our approach, we will be more meticulous in explaining all these steps.

In line 5, we have the *specialize(s)* procedure, that given a clause, in this case a seed s, will produce a specialization of s, in this case $spec_s$. This specialization is such that it takes into account every child and parent of the nodes representing the literals already in the clause and maximizes a given metric.

The reason why we take into account both parents and children of each literal is that so we can search the hypergraph bidirectionally, i.e., we can add a literal to the clause because its input variables are produced by other literals already in the clause or we can add a literal to the clause so a given literal that has an unbounded input variable can remain in the clause with every input variable bounded. Although we do not force the algorithm, in any point, to perform either of these actions, we assume that, since behind the connections of the variables there is potential knowledge, according to [29], we assume that the algorithm, to maximize the chosen metric, will naturally, chose the next literal to add so it contains the most amount of knowledge that better represents the patterns behind the data, i.e., will decide either to introduce a literal so it can produce output variables that are not in the clause or that are in the clause as unbounded input variables on other literals based on what brings the most amount of knowledge and better represents the data. We intend to see all the connections involving the head variables and with this seed generation we anchor, possibly, multiple ends of our search space, just by generating these kind of clauses as seeds. This happens because all the interest variables already exist in our initial hypothesis, then we may, in a very lucky strike, not even have to insert any more literals in the clause. Even if we are not that lucky, this will always be a good starting point, because since we want to relate all of the variables in the bottom clause, a clause that already contains all of those interest variables that we only have to connect using modes is a very useful start.

We exemplify the seed generation process for the hypergraph in Figure 4.1, respecting Example 24, in Figure 4.2.

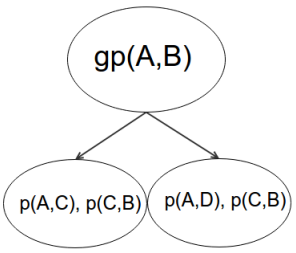


Figure 4.2: Seeds generated for the hypergraph construction in Figure 4.1

Now it only remains to explain the role of the metric in our algorithm. Although the algorithm

can run on every metric, we designed a new, specific metric to enhance our search for concepts that relate all the head variables. Such metric is highly based on the compression metric, that is calculated as $P - N - L$, being P the number of positive examples covered by the clause, N the number of negative examples covered by the clause and L the number of literals (length) of the clause. Our metric is similar, and is calculated using $P - N - PL$ being P the number of positive examples covered by the clause, N the number of negative examples covered by the clause and PL the minimum number of literals so that every variable in the head literal of the target concept are connected through those literals, i.e., it is the path length of the clause, that is the minimum path, using literals, one can draw to relate all head variables in the goal concept. This is the metric used in our system, since it seems to be the one that better distinguishes between the clauses we aim at forming and the rest of the clauses.

More on the metric can be found in the next section, but this is the essential to understand its role in our algorithm. This metric will introduce a good dynamic, since it relates coverage and path length in an interesting way. On one hand we have that clauses have to perform well on coverage, i.e., have to cover, ideally, a good amount of positive examples, and to cover the minimum number of negative examples as possible. On the other hand, this metric also penalizes clauses whose path length measure is high, i.e., whose path that relates all variables in the head clause have many nodes (literals). This introduces the sense that not only we have to represent well the data supported on coverage, but also need to do it with the minimum number of literals possible to avoid having a high penalty with a high PL value. In the light of what we said here, not being able to relate all variables through the literals on the clause, i.e., the non existence of such path, will be penalized with $2 * P$, so that clauses that may not perform so well on the training set but, somehow, found a way of connecting all the head variables, will be of much more interest than those that may have a better performance according to coverage but do not relate all head variables, so not respecting the intuition on [29] that variable connections contain knowledge.

Some conditions form the basis of the justification of some choices we made when designing the algorithm, and those are:

- If all variables are bounded, then the clause will do better than if there are some free variables. This is quite trivial, but it is a good plus for us. Since bounded clauses will be more restrictive, they may exclude some positive examples, since getting a perfect clause may be impossible for a given example, but, using the intuition in [29] that if literals are connected through their variables, then that connection can mean knowledge, we can say that clauses that have all their variables bounded will have more sharing of variables happening, therefore, more knowledge, resulting in better scores for our metric.

- Since we benefit bounded clauses, then we will have more intuitive clauses and easier to read. The fact that these clauses will, somehow, aim at discovering connections between the literals through their variables, will also make these clauses seem like chains of literals that generate each other and carry variables through the literals of the clause. Such reasoning is

very intuitive, and therefore easier to understand, requiring less effort (even from specialists) in accepting such clauses. This is one of the advantages of ILP and this approach can help enhancing it.

The rest of the loop is pretty straightforward, in the sense that if we find a clause that performs better than the best clauses found so far, then we remove all the clauses from the set of best clauses and then introduce this new clause in the set and update the maximum score so far and if we find a clause that performs as better as the best found so far, we only add it to the set of best clauses. Clauses that perform worse than those in the set of clauses that maximize the given metric are not considered anymore.

When considering the seeds in Figure 4.2, the results of the searching algorithm would be the one in Figure 4.3.

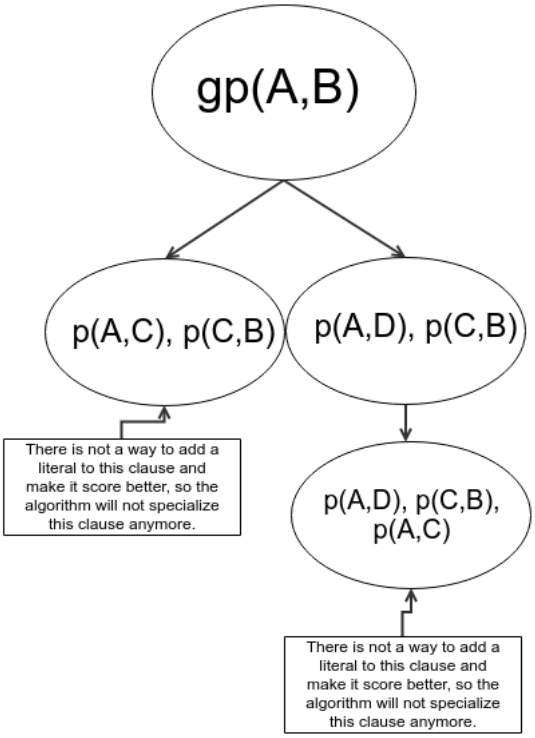


Figure 4.3: Graph describing the generation of new states of clauses for Example 24

Now that all the algorithm is explained, in the next sections we will discuss about some details of the system as well as some nuances that were also integrated into our system that can be introduced to enhance the results of our algorithm.

4.3 OTHER ASPECTS OF THE ALGORITHM

In this section we will talk about some aspects of our system that are not, directly, clearly specified in the algorithm. Such aspects fall on two big subjects: the need for external software and the used heuristic (metric).

4.3.1 External software

In what concerns external software, there is one phase where the access to external software is needed: the saturation phase. To do so, we used an interface to Prolog in our source C++ code, where we connect to Prolog (SWI-Prolog, specifically) and then load and use Aleph to generate the bottom clauses for each analyzed example. There were a pair of reasons to do so. To start, we were familiar with Prolog and Aleph from previous experiences, which meant that there was a familiarity with both of them, being that a really useful factor because of the ability to easily operate with the Aleph system and Prolog in general. Also, the algorithm Aleph uses to saturate a given example was exactly what we needed, in the sense that it was fast enough (it was not a bottleneck in our running time, which can be seen in Chapter 5) and the way to obtain a given bottom clause was relatively easy (only needed to call the *sat* predicate in Aleph followed by the *bottom* predicate).

4.3.2 Metric

The metric used in our algorithm is very important for the type of clauses we want to generate. The metric we used has two main goals: to measure the performance of the algorithm in judging the examples already known and also benefit clauses that may not have the best performance in judging the known examples but have a chain of literals that connect all the head variables. For the first purpose we used coverage and for the second we used path length. So we can calculate how to score a clause using a simple equation $score = P - N - PL$, where P stands for the number of positive examples covered, N stands for the number of negative examples covered and PL stands for the path length of the clause. If there is no path in the clause that connects all the head variables, then we assume $PL = 2 * P$. This value is only so that clauses that do not connect the head variables have a big penalty in their score, when compared to other clauses that may not behave so well in coverage, but make it up by having a lower PL, i.e., by having a path of literals that connects all the head variables.

Although this is a metric made by the authors, based on compression, and that we believe to be the most adequate for the problem and the system's purposes, any other metric could be used when evaluating scores, since the algorithm works with any metric we want. Nonetheless, all our tests used the metric described here, because we were trying to search another search space than the one already browsed by other ILP systems and such search space would be one that prioritizes connected clauses, so this metric would clearly benefit these types of clauses, that

have a chain of literals (a path) that contains all the head variables.

This heuristic has some mutual aspects, but also differs in other ones from the compression measure. The mutual aspect is clear just by looking at both expressions, since our metric is calculated using $P-N-PL$ and compression is calculated using $P-N-L$. Clearly the common aspect is the fact that both of the heuristics make use of the coverage measure ($P-N$), but the last part of the heuristics is clearly different. Such difference occurs when analyzing the penalty both of them apply to the score of a given clause when we have a large L in compression or a large PL in our metric. So compression aims at finding clauses that perform well on the training data, i.e., clauses that have a high coverage value, but also benefits smaller clauses, in the sense that the bigger the clause, the bigger the L and the bigger the penalty in the score of the clause. On the other hand, our metric does not penalize the clause for the number of literals it contains, but instead for the size of the path connecting all the variables in the head of the clause. Although it is not as clear as compression, also leads to the generation of relatively small clauses, since we follow the intuition that the number of literals necessary to explain a given concept is not very high and such literals emerge from overlapping existing variables in the literals of the clause, representing the relations between different literals. So our metric will not benefit clauses with no more than a few literals, but instead clauses with the shortest way, through the literals present in the clause and their variables, to relate all the head variables.

4.4 DIFFERENT STOPPING CRITERIA AND HEURISTICS

Although the standard stopping criterion for our system is to stop when all the positive examples have been analyzed, we propose other two alternatives: only running with a subset of the positive examples and also using cover removal.

Running the algorithm for every positive example in the dataset has its advantages, like analyzing more thoroughly the dataset and softening the effect of noise in the data, since a noisy example will be less relevant in a complete dataset than in a subset of the dataset. On the other hand, it has major disadvantages, like the running time and memory required to run the algorithm. Since the algorithm is exponential because of the combinatorial explosion that arises from analyzing every possible specialization of a clause in any given moment, it is important that we find ways to try to reduce such complexity, even if we continue with an exponential algorithm in theory, we can make it run a little bit faster in practice.

One way to do so is only running the algorithm for a randomly chosen subset of positive examples. Again, some advantages and disadvantages arise when compared to running the algorithm with the complete dataset. In advantages we have the saving in memory spent and running time, because since we sequentially analyze each example of the data set, reducing the number of examples will consume less time and memory than running the algorithm with the complete set of examples. When it comes to disadvantages, we have to have a few precautions when using such stopping criterion. First of all, when choosing randomly the number of examples,

we can be victims of a bias that can, in some cases, affect severely the results of the experiment. To eliminate such bias, one must run the algorithm a few times, always choosing random examples, which can make the memory and time advantages vanish, depending on the cases. Also, there is the fact that noise in the dataset can now have a much higher impact in the results, again because of the bias one is submitted to when running the algorithm with a randomly chosen subset of the set of positive examples.

The third option we have to form our stopping criterion seems to, somehow, combine some of the benefits of using the full set of positive examples with the benefits of choosing just a subset of them. It consists on making an informed and intelligent choice on whether to examine or not a given example. With cover removal, such choice is made by the algorithm by not examining all the examples that are covered by the current theory. This is based on the intuition that if a given theory covers an example, then such example will probably be ruled by the same patterns as the one that originated such theory. So, in practice, analyzing such example would only yield redundant new theories, since the examples describe the same pattern. This is, obviously, not true in every case, since that it is possible that a given example is covered by a given theory, but submitting it to our algorithm could yield a different, and potentially better, one. However, such cases are so specific and so difficult to recreate, normally arising from noisy datasets or datasets with a lot of missing information, that it is used in many ILP systems nowadays. The advantages of using this stopping criterion are the fact that it takes, at most, the same amount of time and memory as running the algorithm with the full set of examples, usually taking considerably less, and it is deterministic, in the sense that running many times the algorithm in the same conditions will not alter the final output, which does not happen when running with a randomly chosen subset of positive examples. Also, it is much resistant to noise than when choosing a random subset of positive examples, since it will discard the examples based on previous knowledge obtained from the dataset, while in a random subset of positive examples one depends on the random choice of examples. However, there are some disadvantages, and the major one is that, although it takes very special conditions and a very high level of it, it is still a little bit sensitive to noise. Also, choosing whether to apply such technique must be considered when taking into account the characteristics of the dataset. For example, if the dataset is unfamiliar, it can be better to first run with every example (if such is possible) or to pre-process the dataset in first instance, before using such technique, since a theory that arises from a noisy example can cover many positive examples that would contain different, potentially more valuable, theories, just because it was analyzed first.

In what concerns different heuristics, although we have a preference for the heuristic described in 4.3.2, because it is the one that fits our purposes better, in the sense that maximizes the chances of obtaining clauses that relate all the head variables, every other heuristic is easily applicable, since it works separately from the algorithm itself.

4.5 DIFFERENCES IN SEARCH SPACE EXPLORED: A PRACTICAL EXAMPLE

Before analyzing an example where we can see clearly the differences between the two algorithms, let us the most important principle of our approach that differ from the existing ones, including Aleph, that being that we simultaneously look at all the paths that arise from all the literals containing variables that are present in the head term. This is evident in the process of generating seeds and in the way of generating the next state of a hypothesis.

To clarify this point, we have an example of the search tree for both algorithms for a given bottom clause.

First, we present the tree for Aleph, in Figure 4.4.

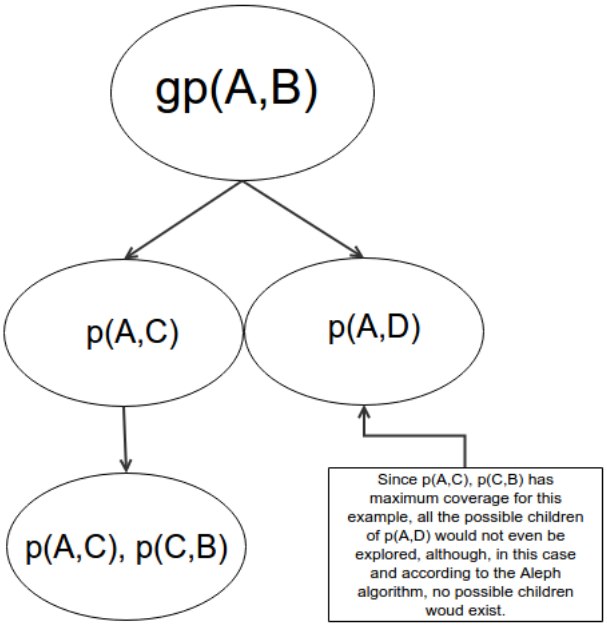


Figure 4.4: Graph describing how Aleph runs the case in Example 24.

The tree for our approach is available in Figure 4.3.

That being said, it is clear that both algorithms will search different parts of the search space, and will find different clauses, since our approach focus on finding connected clauses that have good coverage score, and reinforces this approach with the metric adopted, and Aleph is a more generic algorithm, whose metric can influence the clauses obtained (the system has many possible metric, being coverage the default one).

Since the practical results will only be discussed in 5, we will not, in this chapter, mention anything about the quality of the clauses, but instead, just highlight that, even without measuring

yet the success of our approach and of the clauses generated by it, we can see that different parts of the search space will certainly be checked, in order to find different clauses, that relate the variables of interest and follow the intuition in 3.1.1 to produce clauses that translate the knowledge behind the data presented to the algorithm.

Chapter 5

Materials, Methods and Results

In this chapter, we will present experimental results of our approach and of Aleph for the same datasets. First, we will describe each data set used in testing our work. Then, we will be analyzing the best clauses found and the space searched by each system as well as the time it took to get such clauses and to search through that space. Finally, we will discuss a cross validation test, measuring the average accuracy for each of the approaches in each data set.

5.1 DATA SET DESCRIPTION

In this section we will describe the datasets used to test our approach. We used two datasets: a family relations dataset and the choline dataset. We were hoping to use more than just two datasets, but the difficulty in getting datasets whose specifications would fit our algorithm was a decisive factor in not experimenting our approach in more than two datasets. On the other hand, the two algorithms that were chosen are well known benchmarks in Inductive Logic Programming and the settings of these datasets provide the possibility of us testing our algorithm. Such specifications would be that the goal concept would have to contain more than one variable in the head, the dataset would have to be relatively large and also the concept would have to be a relational one, i.e., concepts like mathematical functions and other related areas are not in the scope of our work, since these kinds of problems were already approached in [32], as mentioned before.

5.1.1 Family relations dataset

This dataset describes parental relationships and was retrieved from the generalogical information of a Croatian family [38]. The relations are of the form *parent(X, Y)* and the task is to learn the grandparent relation. The dataset contains 646 positive examples and the same number of negative examples. The background knowledge is composed of 451 predicates.

5.1.2 Choline dataset

One of the datasets we will be testing in our approach is the famous ILP benchmark choline [14] and it is part of the datasets that relate the activities of drugs used when treating Alzheimer's disease, in this particular case, the dataset related drugs according to their acetocholinesterase inhibition.

The positive and negative examples are pairwise comparisons of drugs and are of the form *great(X,Y)* meaning that X provides a higher acetocholinesterase inhibition than Y.

Background knowledge describes different properties of the chemicals analyzed in the dataset.

The number of positive and negative examples is 663 and the number of predicates in the background knowledge is 647.

5.2 METHODOLOGY

In this section we will describe the general environment of the experiences performed in the two datasets described earlier.

For each dataset, our algorithm was submitted to three scenarios: first, we run the algorithm with the whole dataset, then with 10% of the positive examples chosen randomly and finally we employed cover removal to the whole dataset.

When running the algorithm with the entire dataset, we merely ran our algorithm and Aleph (using the predicate *induce*) for the entire dataset described earlier.

Then, when running a subset of the dataset, that only happened for our algorithm, we ran the experiment 10 times for each dataset, to try to minimize the effect of a potential bias in the random choice of the subset of positive examples.

With cover removal, the setting was similar to the one used when running entire datasets, where we ran our algorithm one time for each dataset.

More details on the some specific settings of the environment used for each dataset (like some Aleph parameters), will be given when discussing in detail the experimental results of each dataset.

We run all tests in a machine with 8 GB of memory and a Intel® Core^{TM} i5-6200U CPU 2.30GHz processor in a 64-bit operation system (Ubuntu 16.04 LTS).

5.3 EXPERIMENTAL RESULTS

In this section we will present the experimental results for each of the datasets explained in 5.1 on both Aleph and our approach. We begin by presenting the results of both approaches in both

datasets, in table 5.1 and their discussion is presented in 5.3.1 and 5.3.2.

Dataset	Family		Choline	
	Aleph	Ours	Aleph	Ours
Runtime (secs)	0.008	4170	8.7	3733
Clauses	5	22356	67988	30553
Max clauses	1	3967	8	859

Table 5.1: Results of both approaches for the family and choline datasets

For clarification reasons, the *Runtime* row presents the number of seconds it took the respective system to solve the problem in the respective dataset, the *Clauses* row represents the number of clauses generated by each system for each dataset and the *Max clauses* field represents the number of clauses returned as the best for each system for a given dataset.

5.3.1 Family relations dataset

The experimental setting for both systems on this dataset is the default for Aleph. Our system uses Aleph for saturation so it inherits many of its characteristics that are not exclusive to the search algorithm itself, so it is possible to also prune the search space in our system changing the settings for Aleph, if those settings affect the generation of the bottom clause.

Running this dataset with our approach yielded the results on Table 5.1 in the respective column.

From the results on Table 5.1 we can see that Aleph is much faster than our approach in generating the rules for this dataset. However, we generate a much larger number of clauses than Aleph. This is explained by the fact that, in this case, Aleph only saturates the first example and, since it gets a clause that covers all positive examples and none of the negative ones, it stops the algorithm and returns the clause *grandparent(X,Y) :- parent(X,Z), parent(Z,Y)*, also stated in 5.2. Our algorithm, on the other hand, does not stop generating new clauses until it has processed every positive example, which means that, in this case, it will process each one of the 646 examples, for completeness purposes, as explained in more detail later. If we take into consideration the average time of generating each clause, the time per clause generated is 0.19 seconds, approximately. The very different number of clauses in our system is explained by the fact that our approach does not exclude unifiable clauses, so two clauses that are exactly the same, since they unify with each other, are still seen as two different clauses by our approach. Although Aleph already employs a mechanism to remove redundant rules, we still have not implemented such mechanism because of some limitations with the Prolog interface to C++, but this is part of a future work for this approach. However, after a meticulous analysis, we could see that all of the clauses returned by our approach fall into two different types of clauses, as stated in 5.3. Both of those clauses unify with *grandparent(X,Y) :- parent(X,Z), parent(Z,Y)* and this is known to be the correct definition of grandparent and also the same as Aleph.

Clause output by Aleph	Coverage
grandparent(X,Y):-parent(X,Z),parent(Z,Y)	646

Table 5.2: Clause returned by Aleph when running the family dataset

Clauses output by our approach	Coverage
grandparent(X,Y):-parent(X,Z),parent(Z,Y)	646
grandparent(X,Y) :- parent(X,K), parent(Z,Y), parent(X,Z)	646

Table 5.3: Clauses returned by our approach when running the family dataset

Also, and looking at Figures 5.1 and 5.2 and Tables 5.4 and 5.5, of the running time for our approach we have to refer that 15.7 seconds were spent saturating every example, 9.46 seconds creating the hypergraph corresponding to the bottom clause and 4074.89 seconds were spent on the search algorithm itself, and of those 4074.89 seconds, 3410.86 were spent calculating the coverage of each candidate clause and 275.61 were spent calculating the path length of each candidate clause.

Operation	Time spent	Percentage of total time spent
Running Time	4170	100%
Saturation	15.7	0,38%
Hypergraph Creation	9,46	0,23%
Search algorithm	4074.89	97,72%

Table 5.4: Table representing how the runtime divided itself into saturating clauses, creating hypergraphs and on the search algorithm for the family dataset

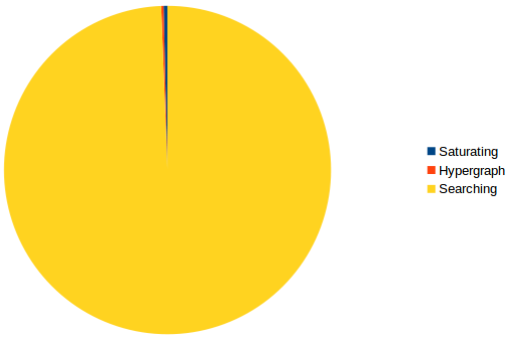


Figure 5.1: Pie chart representing how the runtime divided itself into saturating clauses, creating hypergraphs and on the search algorithm for the family dataset

Operation	Time spent	Percentage of total time spent
Search Algorithm	4074.89	100%
Coverage	3410.86	83,7%
Path Length	275.61	6,76%

Table 5.5: Table representing how the runtime of the search algorithm divided itself into calculating coverage and path length for the family dataset

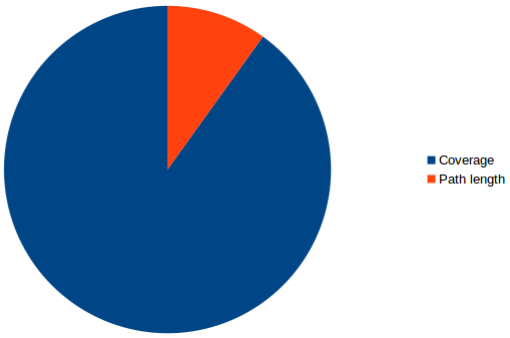


Figure 5.2: Pie chart representing how the runtime of the search algorithm divided itself into calculating coverage and path length for the family dataset

In Table 5.6 we have the result on running our algorithm only on 10% of the positive examples of the data set, chosen randomly. The results show, obviously, a much smaller runtime, since we are only running the algorithm with a small subset of the examples (64, precisely). However, as seen in Table 5.7, the best clauses returned by applying our approach to a subset of 10% of the total set of examples are exactly the same as running the algorithm with the complete set of positive examples. This means that, in datasets whose data is consistent enough, in the sense that the data has its patterns well distributed along all the examples, containing minimum values of noise, it is possibly a good alternative to apply the algorithm to a subset of the set of examples, obtaining similar or equal results if the quality of the data set is high enough. Although a bit of randomness is always introduced when choosing, yet again, random examples, these tests were performed 10 times and, at every time, 10% of the examples were randomly chosen.

Running time	Nº clauses	Nº max clauses
498.31	1759	381

Table 5.6: Results of our approach when running the family dataset for 10% of the positive examples

Clauses output by our approach	Coverage
grandparent(X,Y):-parent(X,Z),parent(Z,Y)	646
grandparent(X,Y) :- parent(X,K), parent(Z,Y), parent(X,Z)	646

Table 5.7: Clauses returned by our approach when running the family dataset for 10% of the positive examples

Another way we tested our algorithm was choosing the next positive example to saturate according to cover removal, i.e., for each new theory that has a score equal or higher than the current maximum score, the positive examples that this theory covers are not submitted to our algorithm. This change to the algorithm follows the intuition that if a given clause is extracted from the information of a positive example and there is another different positive example covered by that clause, then they share information and it is very likely that applying the same algorithm to that other example will generate a redundant clause. Note that it is very likely to happen, and not certain, since an example can be covered by a clause but, when submitted to our approach, yield a different clause.

Adding cover removal to our approach will lower the runtime, but, instead of choosing random examples, as in the previous discussion, the algorithm will now make smarter choices of which examples it will work with, discarding the ones that, probably, will not bring new information to the table. Since cover removal, the way we implemented it, is deterministic (which is not true for every system), since it follows the order of the positive examples as in the file, the results will not vary when done repeatedly, which may not happen when choosing randomly a subset of positive examples.

So, after applying our algorithm with cover removal, we got the results in 5.8 and 5.9. In what concerns running time, we have a much smaller value because only one example is saturated and analyzed, since the theory extracted from it, as it would happen for any example in this dataset, covers all the positive examples. The extracted clauses are exactly the same and only three clauses are generated. This means that cover removal has the ability to maintain the quality of the clauses extracted from a given dataset, although it reduces the memory spent to accomplish it (only generated three clauses) and the runtime, if the dataset is consistent, in the sense that most of the positive examples follow the same pattern, and clean enough, in the sense that noise and missing values are not an issue, to be applied to cover removal, otherwise, maybe pre-processing the dataset or applying it to every positive example in first instance may be the best choice.

Running time	Nº clauses	Nº max clauses
1.5	3	2

Table 5.8: Results of our approach when running the family dataset with cover removal

Clause output by our approach with cover removal	Coverage
grandparent(X,Y):-parent(X,Z),parent(Z,Y)	646
grandparent(X,Y) :- parent(X,K), parent(Z,Y), parent(X,Z)	646

Table 5.9: Clause returned by our approach when running the family dataset with cover removal

5.3.2 Choline dataset

The experimental setting for both systems on this dataset is the default for Aleph except on the *nodes*, *i* and *clauselength* settings that are set to 1000000, 2 and 3. Also, the noise is set to 600, since we want Aleph to search for clauses that cover some negative examples, since our system also does it and it is not possible to find a clause that covers every positive example and none of the negative positives in this data set, but this setting does not affect our algorithm, since the noise setting does not affect the saturation of the examples.

Running this dataset with our approach yielded the results on Table 5.1.

From the results on Table 5.1, we can see, again, that Aleph is much faster than our approach, with a similar justification to the one given in 5.3.1. Also, the average time per clause generated is 0,12 seconds. From the 44 clauses yielded by Aleph only 8 are not ground facts (that only cover 1 positive example and none of the negatives). Those 8 clauses are presented in table 5.10. Also, most of these rules also cover negative examples. We also want to highlight the fact that none of these rules relates all head variables and sometimes the second variable does not appear in the clause at all, one of the problems our system tries to solve. Our approach on the other hand outputs 859 clauses but all of them unify either with one of the clauses in Table 5.11.

Clauses output by Aleph	Coverage
great(A,B) :- alk_groups(B,4), alk_groups(A,0)	21
great(A,B) :- ring_subst_3(B,C), n_val(A,D)	7
great(A,B) :- alk_groups(B,4), alk_groups(A,3)	6
great(A,B) :- alk_groups(A,2), ring_subst_3(B,C)	14
great(A,B) :- alk_groups(B,4), alk_groups(A,2)	6
great(A,B) :- alk_groups(B,4), ring_subst_4(B,C)	36
great(A,B) :- alk_groups(B,4), ring_subst_2(A,C)	21
great(A,B) :- ring_substitutions(A,5), ring_subst_3(B,C)	6

Table 5.10: Clauses returned by Aleph when running the choline dataset

Clauses output by our approach	Coverage
great(A,B):-r_subst_1(A,C),r_subst_2(B,D)	158
great(A,B):- alk_groups(A,C),r_subst_2(B,D)	158

Table 5.11: Clauses returned by our approach when running the choline dataset

In our approach, we always start the search choosing literals from the bottom clause that

contain the concept variables. For this dataset, in particular, r_subst_1/2 and r_subst_2/2 and alk_groups/2 are three literals that are used in the roots of our search trees. As no refinement improves this set of literals, our algorithm outputs the result without performing further searches.

Also, as one can see in Figures 5.3 and 5.4 and Tables 5.12 and 5.13, of the running time for our approach we have to refer that only 15.98 seconds were spent saturating every example, 38.41 seconds creating the hypergraph corresponding to the bottom clause and 3614.23 seconds were spent on the search algorithm itself, and of those 3614.23 seconds, 2769.07 were spent calculating the coverage of each candidate clause and 715.43 were spent calculating the path length of each candidate clause.

Operation	**Time spent**	**Percentage of total time spent**
Running Time	3733	100%
Saturation	15.98	0,43%
Hypergraph Creation	38.41	1,03%
Search algorithm	3614.23	96,82%

Table 5.12: Table representing how the runtime divided itself into saturating clauses, creating hypergraphs and on the search algorithm for the choline dataset

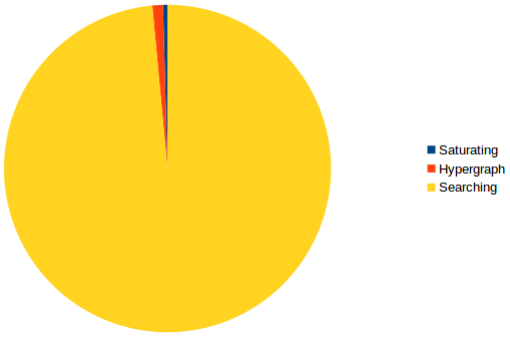


Figure 5.3: Pie chart representing how the runtime divided itself into saturating clauses, creating hypergraphs and on the search algorithm for the choline dataset

Operation	**Time spent**	**Percentage of total time spent**
Search Algorithm	3614.23	100%
Coverage	2769.07	76,62%
Path Length	715.43	19,79%

Table 5.13: Table representing how the runtime of the search algorithm divided itself into calculating coverage and path length for the family dataset

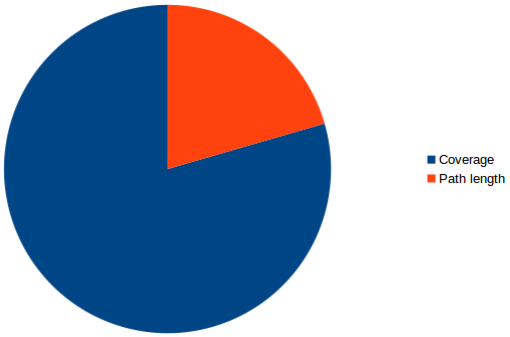


Figure 5.4: Pie chart representing how the runtime of the search algorithm divided itself into calculating coverage and path length for the choline dataset

In this case we also tested our approach to only 10% of the positive examples.

When running the algorithm to the complete set of positive examples, we stated that a large number of positives yielded the same theory. This made us suspect that perhaps running the algorithm with the full set of examples could be, at some point, somewhat redundant. So, as we did in the family dataset, we decided to apply our approach only to 10% of the positive examples. The results, as it happened in the family dataset for the identical test, available at 5.14 and 5.15, were, once again, the same as running the algorithm with the full set of positives, in terms of the clauses outputted, but, the total number of clauses generated and the runtime were significantly smaller. This confirms that, when the dataset is consistent and does not have much noise, not containing much noise or missing values, as it is the case of this dataset and the family dataset, just running the algorithm to a subset of the total examples, can also yield good results, sometimes as good as running the algorithm to the full set, but taking significantly less time.

Running time	**Nº clauses**	**Nº max clauses**
527.4	2912	85

Table 5.14: Results of our approach when running the choline dataset for 10% of the positive examples

Clauses output by our approach	**Coverage**
great(A,B):-r_subst_1(A,C),r_subst_2(B,D)	158
great(A,B):- alk_groups(A,C),r_subst_2(B,D)	158

Table 5.15: Clauses returned by our approach when running the choline dataset for 10% of the positive examples

As we did for the family dataset, we also applied the modified version of our algorithm with cover removal to the choline dataset, whose results are in Table 5.16 and Table 5.17. With this enhancement in choosing which positive examples will be analyzed and which will not, we were able to reduce significantly the memory spent, since less clauses were generated, as well as the

Running time	Nº clauses	Nº max clauses
652.7	1683	2

Table 5.16: Results of our approach when running the family dataset with cover removal

Clause output by our approach with cover removal	Coverage
great(A,B):-r_subst_1(A,C),r_subst_2(B,D)	
great(A,B):- alk_groups(A,C),r_subst_2(B,D)	

Table 5.17: Clause returned by our approach when running the family dataset with cover removal

execution time, maintaining the quality of the best clauses encountered, being those the same as in the previous experiments for this dataset. Although in this case the time is still significantly higher than the one taken by Aleph to analyze the same dataset, it is still much smaller than the one taken by our approach to process the full set of examples for the choline dataset and the quality stays untouched.

5.3.3 General conclusions

With the results discussed in the previous sections we can see that our goals were accomplished, since the number of generated clauses is way bigger and the returned clauses are different, meaning that the hypothesis space searched by our algorithm is not the same as the one searched by Aleph. Also, although the running time of our approach is way bigger, it is justifiable by the large number of clauses it generates, mainly because of calculating the coverage metric for each candidate clause, that takes most of the time spent in the search problem. The quality of our solutions is better than the one from the solutions of Aleph, in the choline dataset specially, since that, in the family one both systems yield a clause that covers all positive examples and zero negatives, but it is not a great improvement, at least looking at coverage only, since the clauses returned by both systems are generally bad, covering (1) only a small part of the positives, or (2) a good part of the positives, but also a very big part of the negatives. Concluding, without having, yet, analyzed, in more detail, the accuracy of both systems (it will be done through cross validation 5.3.4), we consider interesting that we could achieve results similar to Aleph, in what concerns coverage, when searching the hypothesis space in a much different way. Also, some future adjustments that will be done to the algorithm will, certainly, enhance its major flaws: the excessive running time and the returned clauses that are unifiable with each other.

5.3.4 Cross validation

In k-fold cross validation, we divide the dataset D randomly into k exclusive mutually subsets (folds) $D_1, \ldots D_k$. We train the algorithm and test it k times and each time $t \in \{1, \ldots, k\}$, we train the system on $D - D_t$ and test it in D_t. The cross-validation estimate of accuracy is the overall number of correct classifications divided by the number of instances in the dataset. So we

System	Acc 1	Acc 2	Acc 3	Acc 4	Acc 5	Average
Aleph	1	1	1	1	1	1
Our Approach	1	1	1	1	1	1

Table 5.18: Results on both approaches for cross validation on the family dataset

System	Acc 1	Acc 2	Acc 3	Acc 4	Acc 5	Average
Aleph	0.5094	0.5097	0.5101	0.5095	0.5097	0.5097
Our Approach	.614	.607	.598	.621	.611	.610

Table 5.19: Results on both approaches for cross validation on the choline dataset

define $accuracy = \frac{TP+TN}{TP+TN+FP+FN}$, being TP the true positives, TN the true negatives, FP the false positives and FN the false negatives. Also, for stability purposes, i.e., to make sure that the results of cross validation presented do not vary with the random fold generation, we ran cross validation 5 times per dataset, and since the results were stable, there was no need to keep performing cross validation on the datasets. This method is used to get an idea of the quality of the rules generated by our algorithm, in the sense that it calculated the accuracy of those rules in the test fold.

5.3.4.1 Family relations dataset

For the family dataset, after running cross validation and Aleph in the same running conditions as in 5.3.1, the following results were yielded in each iteration 5.18.

All the fields $Acci$, $i \in \{1, \ldots, 5\}$ represent the average accuracy of the hypothesis for each iteration of the 5 time 10-fold cross validation evaluation process. The last field (*Average*) represents the average of the 5 iterations. As expected by running both algorithms just once for the family dataset, in every iteration, the average accuracy of the outputted clauses if 1, i.e., the clauses from both systems classify correctly all of the examples in the dataset.

5.3.4.2 Choline dataset

For the choline dataset, after running cross validation and Aleph in the same running conditions as in Section 5.3.2, the following results were yielded in each iteration in Table 5.19.

All the fields $Acci$, $i \in \{1, \ldots, 5\}$ represent the average accuracy of the hypothesis for each iteration of the 5 time 10-fold cross validation evaluation process. The last field (*Average*) represents the average of the 5 iterations.

The results for accuracy in Aleph and in out approach were obtained by averaging the accuracy in the unseen fold of each of the returned rules that formed the final theory, except the ground facts Aleph returned as rules. Since the variation of the results was minimal in these five iterations of the cross validation algorithm, we thought the number of iterations was enough to

make considerations about the quality of the clauses produced by the two systems. We can see that our approach performs a little bit better than Aleph for this dataset, although neither of the systems perform very well, since the accuracy value is not very high. Even though our approach only returns the seed clauses in every iteration of the cross validation algorithm, they seem to perform better for unseen data, according to accuracy, than the clauses returned by Aleph, that, like stated in 5.3.2, do not even contain all the head variables in most cases (connected by literals or not).

Also, most of the rules generated by Aleph were simply ground facts, and those that were not ground facts, did not behave very well in unseen cases.

Analyzing the statistical significance of the results, we get that $p(T < t) = 0,004$, for the statistical t-test, performed using *LibreOffice Calc*, meaning that the results are statistical significant, in the sense that $p(T < t) < 0,05$, being that the generally threshold considered when analyzing results like these. One might argue that running an algorithm for many hours just to get an increase of 0.1 in accuracy in average may not be a good tradeoff. However, in certain domains, like the medical field for instance, errors can be very penalized, so whether the algorithm is suitable for a given problem depends a little bit on the problem itself, when comparing to other alternatives, in this case Aleph.

5.4 FINAL THOUGHTS

Analyzing the experimental results for our work, we can see that it has some pros and cons when compared to Aleph, but also we consider it has its space in the ILP paradigm and this work is a good starting point for promising future work that will, possibly, result in better algorithms and systems than the existing ones for solving relation machine learning problems.

When compared to Aleph, our approach takes much more time to finish the task, which, depending on the problem, can be a relatively big issue. However, this is the only big flaw of our work, when compared to Aleph, in our opinion. The quality of the clauses, when it comes to accuracy, is equal or even better in the datasets used to perform the evaluation of both systems, which is a good indicator that our approach can produce good quality clauses. Also, even though the time taken to finish the same task is way bigger in our algorithm, we also search a much bigger search space, since we are searching the hypothesis space in a different manner, trying to connect and relate all head variables.

Concluding, searching the hypothesis space in the way we suggest seems to yield good results on relational datasets, when we look at the experimental results, namely the quality of the theories.

Chapter 6

Conclusions

In this thesis we were able to construct an ILP system that searches the hypothesis space benefiting clauses that relate the head variables, taking into account the intuition that if there is a connection between the head variables, that connection, most probably, contains knowledge.

After building such algorithm and consequent ILP system, the experimental results showed that our approach yields better results than Aleph, in the tested datasets, which is a good indicator that the clauses generated by our system have good quality.

However, many difficulties were surpassed during the implementation of our ideas. First, the algorithm itself suffered many adjusts through the months we worked on it. Then, the SWI-Prolog interface does not have a very detailed documentation, which was a problem in many steps of our algorithm. Finally, getting datasets that were suitable for the problem we were trying to solve was not an easy task, as well.

Nonetheless, it was with great joy that we presented this system and with big hope that this is just the beginning of a great idea.

6.1 FUTURE WORK

In the future, some work can be done in order to enhance the proposed algorithm.

First, since most of the running time of the algorithm is spent on coverage, work can be done to improve the way the algorithm performs coverage [17], since that, currently, when we evaluate the coverage of a given clause, we check its coverage for every example in the dataset. Since adding a literal to a new clause will only cover, at best, the same number of examples that the previous clause covered, that information can be used to enlarge the bottleneck stated in the running time of the algorithm related to the time spent on calculating the coverage of every candidate clause and future children clauses of it.

Also, a lot of the clauses returned are unifiable with each other, which means that implementing

some kind of unification in our system would enhance not only the final output (that would contain only non unifiable clauses), but also unifiable bottom clauses would not be all explored, since exploring all of them will generate the same results than exploring just one of them.

Since our algorithm is able to process a lot of data, a future parallel version of our algorithm could yield good results.

To finish the future work suggestions, applying our algorithm to datasets with more than two head variables and study such application would be very interesting.

Bibliography

[1] Kamal M Ali and Michael J Pazzani. Hydra: A noise-tolerant relational concept learning algorithm. In *IJCAI*, pages 1064–1071. Elsevier, 1993.

[2] L.M. Augusto. *Logical Consequences: Theory and Applications: An Introduction*. Studies in logic. College Publications, 2017. ISBN: 9781848902367.

[3] Giorgio Ausiellоyz, Roberto Giaccioy, Giuseppe F Italianox, and Umberto Nanniyz. Optimal traversal of directed hypergraphs. *ICSI, Berkeley, CA*, 1992.

[4] Hendrik Blockeel and Luc De Raedt. Top-down induction of first-order logical decision trees. *Artificial intelligence*, 101(1-2):285–297, 1998.

[5] Ivan Bratko. Applications of machine learning: Towards knowledge synthesis. *New Generation Computing*, 11(3):343–360, 1993.

[6] Elizabeth S Burnside, Jesse Davis, Vítor Santos Costa, Inês de Castro Dutra, Charles E Kahn Jr, Jason Fine, and David Page. Knowledge discovery from structured mammography reports using inductive logic programming. In *AMIA Annual Symposium Proceedings*, volume 2005, page 96. American Medical Informatics Association, 2005.

[7] Keith L Clark. Negation as failure. In *Logic and data bases*, pages 293–322. Springer, 1978.

[8] Saso Dzeroski and Ivan Bratko. Handling noise in inductive logic programming. In *Proceedings of the International Workshop on Inductive Logic Programming*, volume 91, 1992.

[9] Sašo Dzeroski and Nada Lavrac. Inductive logic programming: Techniques and applications, 1994.

[10] Jean Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving*. Dover, 2015, 2003.

[11] Alfred Horn. On sentences which are true of direct unions of algebras. *The Journal of Symbolic Logic*, 16(1):14–21, 1951.

[12] Jörg-Uwe Kietz and Stefan Wrobel. Controlling the complexity of learning in logic through syntactic and task-oriented models. In *Inductive logic programming*. Citeseer, 1992.

[13] R. King, S. Muggleton, and M. Sternberg. Predicting protein secondary structure using inductive logic programming. *Protein Engineering*, 5:647–657, 1992.

[14] Ross D King, Michael JE Sternberg, and Ashwin Srinivasan. Relating chemical activity to structure: an examination of ilp successes. *New Generation Computing*, 13(3):411–433, 1995.

[15] Y Kodratoff, D Sleeman, M Uszynski, K Causse, and S Craw. Building a machine learning toolbox. *Enhancing the Knowledge Engineering Process*, pages 81–108, 1992.

[16] Nada Lavrač, Sašo Džeroski, and Marko Grobelnik. Learning nonrecursive definitions of relations with linus. In *Machine learning—EWSL-91*, pages 265–281. Springer, 1991.

[17] Carlos Alberto Martínez-Angeles, Haicheng Wu, Inês Dutra, Vítor Santos Costa, and Jorge Buenabad-Chávez. Relational learning with gpus: accelerating rule coverage. *International Journal of Parallel Programming*, 44(3):663–685, 2016.

[18] Eric Kao Michael Genesereth. Herbrand semantics, 2010.

[19] Donald Michie. Machine learning in the next five years. In *Proceedings of the 3rd European Conference on European Working Session on Learning*, pages 107–122. Pitman Publishing, 1988.

[20] Tom M Mitchell. Generalization as search. *Artificial intelligence*, 18(2):203–226, 1982.

[21] Stephen Muggleton. Inductive logic programming. *New generation computing*, 8(4):295–318, 1991.

[22] Stephen Muggleton. Inverse entailment and progol. *New generation computing*, 13(3): 245–286, 1995.

[23] Stephen Muggleton and Wray Buntine. Machine invention of first order predicates by inverting resolution. In *Proceedings of the Fifth International Conference on Machine Learning*, pages 339–352, 12 1988.

[24] Stephen Muggleton, Cao Feng, et al. *Efficient induction of logic programs*. Turing Institute, 1990.

[25] Stephen Muggleton, Ross D King, and Michael JE Stenberg. Protein secondary structure prediction using logic-based machine learning. *Protein Engineering*, 5(7):647–657, 1992.

[26] Irene M Ong, Inês de Castro Dutra, David Page, and Vítor Santos Costa. Mode directed path finding. In *European Conference on Machine Learning*, pages 673–681. Springer, 2005.

[27] Michael Pazzani and Dennis Kibler. The utility of knowledge in inductive learning. *Machine learning*, 9(1):57–94, 1992.

[28] J. Ross Quinlan. Learning logical definitions from relations. *Machine learning*, 5(3):239–266, 1990.

[29] Bradley L Richards and Raymond J Mooney. *Learning relations by pathfinding*. Artificial Intelligence Laboratory, University of Texas at Austin, 1992.

[30] Bradley L Richards and Raymond J Mooney. Automated refinement of first-order horn-clause domain theories. *Machine Learning*, 19(2):95–131, 1995.

[31] J. A. Robinson. Logic and logic programming. *Commun. ACM*, 35(3):40–65, March 1992. ISSN: 0001-0782. doi:10.1145/131295.131296.

[32] José CA Santos, Alireza Tamaddoni-Nezhad, and Stephen Muggleton. An ilp system for learning head output connected predicates. In *Portuguese Conference on Artificial Intelligence*, pages 150–159. Springer, 2009.

[33] George F Schueler. Modus ponens and moral realism. *Ethics*, 98(3):492–500, 1988.

[34] Ehud Y Shapiro. *Algorithmic program debugging*. MIT press, 1983.

[35] Ashwin Srinivasan. Aleph - a learning engine for proposing hypothesis, mar 2007.

[36] Maarten H Van Emden and Robert A Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM (JACM)*, 23(4):733–742, 1976.

[37] Ruediger Wirth and Paul O'Rorke. Constraints on predicate invention. In *Proceedings of the 8th International Workshop on Machine Learning*, pages 457–461, 1991.

[38] Vladimir Batagelj Wouter de Nooy, Andrej Mrvar. Exploratory social network analysis with pajek, 2004.

[39] Wojciech Ziarko and Ning Shan. A method for computing all maximally general rules in attribute-value systems. *Computational Intelligence*, 12(2):223–234, 1996.